UML

METODOLOGIAS E FERRAMENTAS

CASE

CENTROATLANTICO.PT

Alberto Silva
Carlos Videira

Alberto Manuel Rodrigues da Silva
Carlos Alberto Escaleira Videira

UML, Metodologias e Ferramentas CASE

Linguagem de Modelação UML, Metodologias e Ferramentas CASE na Concepção e Desenvolvimento de Software

Edições Centro Atlântico
Portugal/2001

À Graça, Joana e João Alberto

Alberto Silva

À Elsa, Sofia e Guilherme

Carlos Videira

Peça, gratuitamente, os ficheiros com as soluções dos exercícios ímpares deste livro

Receba gratuitamente, por e-mail, os ficheiros com as soluções dos exercícios ímpares deste livro, para poder comparar com as suas respostas. Para tal, envie a cópia da factura deste livro para o Centro Atlântico, para o e-mail,

geral@centroatlantico.pt

ou por correio para,

Centro Atlântico
Ap. 413
4760 V. N. Famalicão

Prefácio

Objectivos, Contexto e Motivação

O livro “UML, Metodologias e Ferramentas CASE” aborda tópicos importantes para a generalidade dos intervenientes nas actividades enquadradas na engenharia de software, designadamente as problemáticas (1) das linguagens de modelação de software, (2) do processo e das metodologias de desenvolvimento de software, e (3) das ferramentas CASE de suporte à modelação e ao próprio desenvolvimento. Pretende dar uma panorâmica abrangente sobre estes três aspectos de forma integrada e coerente. Embora o foco do livro seja nas fases de concepção de sistemas de software, discute o seu enquadramento de modo mais lato em áreas como o planeamento estratégico de sistemas de informação; as arquitecturas de sistemas de informação; ou mesmo a engenharia de software.

O livro explica a necessidade da modelação no desenvolvimento de software, o que é o UML (*Unified Modeling Language*), como aplicar o UML no contexto mais abrangente das metodologias e processos de desenvolvimento, e como usar ferramentas CASE de forma a maximizar e automatizar algumas das tarefas relacionadas com a modelação, por exemplo, produção e gestão de documentação, geração de código, geração de esquemas de dados, *reverse engineering*, *round-trip engineering*, mecanismos de extensão, etc.

A aprendizagem e adopção dos temas abordados neste livro constituem uma vantagem decisiva para os intervenientes que os adoptarem consistentemente. Entre outros, salientamos os seguintes benefícios: melhor documentação dos sistemas e dos respectivos artefactos; aplicação de técnicas de modelação orientadas por objectos, mais fáceis de entender; reutilização desde as fases preliminares da concepção até à implementação; rastreabilidade dos requisitos ao longo de todo o processo; facilidade de comunicação entre todos os intervenientes envolvidos

no processo; melhorias significativas em factores como sejam flexibilidade e produtividade; melhor gestão de requisitos; avaliação e manutenção de sistemas mais facilitadas. Estas características são naturalmente interdependentes entre si; por exemplo, uma maior qualidade da documentação produzida possibilita uma melhor comunicação entre os intervenientes de um projecto, ou uma melhor manutenção entre eles.

Todavia, os assuntos tratados neste livro são difíceis de adoptar nas organizações, por inúmeras razões. Antes de mais porque o ritmo de inovação tecnológica nesta área da engenharia tem-se processado a um ritmo particularmente intenso.

A segunda razão deve-se ao facto dos tópicos abordados neste livro exigirem uma formação significativa e principalmente uma adequada e correspondente actuação. Não basta dominar um conjunto alargado de conceitos e notações para especificar software, mas é fundamental aprender a aplicá-los de forma consistente, repetida e sistemática; adaptá-los às condicionantes e realidades de cada empresa, ou de cada projecto em particular; e ainda partilhar técnicas e métodos entre todos os indivíduos da empresa, ou de cada projecto, para que a comunicação entre todos os intervenientes seja maximizada e eficiente.

A terceira razão, consequência das razões anteriormente referidas, é o facto de ser oneroso a adopção efectiva e produtiva (dos tópicos abordados neste livro) no seio das empresas. Oneroso em termos do tempo inicial que é necessário despender em formação, em termos da "resistência à mudança", assim como o investimento necessário na selecção e aquisição de ferramentas CASE que potenciem significativamente as suas vantagens.

Este livro surge na sequência da experiência dos autores em actividades de investigação, mas principalmente em actividades de consultoria e de docência nas áreas de engenharia de software e de sistemas de informação.

Os temas abordados neste livro são na sua maioria influenciados pelo trabalho de unificação e de evangelização dos "três amigos": Grady Booch, Ivar Jacobson e James Rumbaugh. Todavia, é da nossa exclusiva responsabilidade o estilo do livro, assim como a sua estrutura, conteúdo, exemplos e exercícios propostos (tal como as correspondentes

gralhas e omissões decorrentes!). O livro condensa e integra informação dispersa por alguns livros da área, em particular os seguintes títulos: *OMG Unified Modeling Language Specification* [OMG99], *The Unified Modeling Language User Guide* [Booch99], *The Unified Software Development Process* [Jacobson99], *Visual Modeling with Rational Rose 2000 and UML* [Quatrani00] e *The Rational Unified Process* [Kruchten00]. No entanto, há inúmeros aspectos que o livro propõe e discute de forma única, dificilmente encontrados em qualquer dos livros referidos.

A nível internacional, existe um número relevante de títulos nesta área; contudo, há reconhecidamente na língua Portuguesa uma lacuna muito significativa. Paralelamente, e em consequência da nossa experiência e responsabilidade de docência, supervisão e coordenação de trabalhos finais de curso e de investigação identificámos a necessidade e oportunidade de produzirmos este livro com vista a apoiar a aprendizagem da engenharia de software nos tópicos referidos.

A temática tratada neste livro é abrangente e a sua profundidade é, propositadamente, de nível intermédio. Ínumeros assuntos poderão ser analisados e aprofundados complementarmente, entre os quais se destacam a título de exemplo os seguintes: arquitecturas de sistemas de software [Hofmeister99]; processos de negócio em contextos organizacionais [Penker00]; padrões de análise [Fowler96]; padrões de desenho em infra-estruturas de software (*frameworks*) [Souza99]; modelação de dados [Muller00]; modelação de aplicações segundo o paradigma dos agentes de software [Odell00], modelação de aplicações de tempo real [Selic94], ou modelação de aplicações interactivas [Nunes99]. Todos estes tópicos são importantes nos seus respectivos contextos de aplicação; muitos são alvo de intensa actividade de estudo e investigação. Todos eles apresentam, contudo, um denominador comum: baseiam-se no conhecimento introduzido, apresentado e discutido neste livro.

Audiência do Livro

O livro pretende servir como referência de suporte a um número restrito de disciplinas de nível de ensino superior na área de sistemas de informação. Consequentemente, o livro adopta um estilo tendencialmente pedagógico através da apresentação e discussão de exemplos, da narrativa de histórias e factos reais, ou pela proposta de exercícios académicos.

O primeiro perfil de leitores deste livro vai directamente para os alunos de licenciatura e de cursos de pós-graduação em engenharia informática ou em informática de gestão. Pressupõe-se que os leitores já "assbem" implementar aplicações informáticas; e que neste livro procuram aprender a reflectir sobre o processo de desenvolvimento de software, e aprender técnicas e práticas consistentes e sistemáticas para o realizar.

Adicionalmente, este livro é relevante para um número mais alargado de leitores, em particular para investigadores, gestores informáticos, responsáveis pelo processo de desenvolvimento de software, analistas-programadores, e outros que necessitem de especificar de forma mais ou menos detalhada sistemas de software.

O livro pressupõe um conjunto de pré-requisitos que o leitor deverá possuir para o poder usufruir devidamente. É suposto o leitor possuir um conhecimento razoável sobre as bases da informática e dos sistemas de computadores, tais como noções essenciais de programação, de bases de dados e de sistemas operativos.

Organização do Livro

O livro encontra-se organizado em 4 partes, 14 capítulos e 2 apêndices conforme se resume de seguida.

A Parte 1 (INTRODUÇÃO E VISÃO GERAL) apresenta os conceitos gerais, visão histórica e enquadramento da realização deste livro. Inclui os capítulos 1, 2 e 3.

A Parte 2 (LINGUAGEM DE MODELAÇÃO UML) é constituída por 6 capítulos complementares, sendo que o Capítulo 4 dá a visão histórica e geral do UML e o Capítulo 9 descreve sucintamente alguns aspectos

considerados “avançados”, não essenciais para o leitor que apenas pretende usar e aplicar as características básicas do UML. Os restantes capítulos (Capítulos 5, 6, 7 e 8) constituem o centro desta parte do livro e deverão ser lidos de forma sequencial conforme proposto.

A Parte 3 (METODOLOGIAS DE DESENVOLVIMENTO DE SOFTWARE) apresenta a problemática geral das metodologias e processos de desenvolvimento de software, com exemplos concretos baseados em duas propostas reais de metodologias, o RUP e o ICONIX, descritos respectivamente nos Capítulos 10 a 11.

A Parte 4 (FERRAMENTAS CASE) apresenta a problemática das ferramentas CASE descrevendo o seu significado, evolução histórica e discutindo mecanismos de caracterização e avaliação (Capítulo 12). São apresentadas e analisadas duas ferramentas CASE, o Rose da Rational e o System Architect da Popkin, respectivamente nos Capítulos 13 e 14.

No Apêndice A (“Guia de Recursos Electrónicos”) apresenta-se de modo classificado um conjunto significativo de recursos electrónicos sobre os temas abordados neste livro.

No Apêndice B (“Glossário, Siglas e Abreviaturas”) apresentam-se três tabelas com informação relativa ao glossário, as siglas, e as abreviaturas adoptadas ao longo de todo o livro.

Em “Referências” listam-se, por ordem alfabética, todas as referências bibliográficas utilizadas ao longo do livro.

Por fim, inclui-se o “Índice Remissivo” que constitui um mecanismo típico de trabalho e de consulta neste género de literatura.

Notação Adoptada

Ao longo do livro são adoptadas genericamente as seguintes regras de notação textual:

- Nomes e expressões em inglês são escritas em itálico. As excepções são expressões vulgarmente adoptadas para o Português (e.g., software, bit), expressões intensamente usadas ao longo do texto (e.g., Internet, Web, applet, standard), ou nomes de empresas ou produtos de origem anglo-saxónica (e.g., MS-Word, Rational Rose).
- Frases e expressões que se pretendam destacar são escritas com ênfase (i.e., negrito).
- Exemplos de código, pseudo código, nomes de classes, ou endereços electrónicos são apresentados numa fonte de tamanho fixo (i.e., `Courier`).

Os exemplos apresentados neste livro aparecem enquadrados por uma moldura correspondente, conforme ilustrado neste mesmo parágrafo.

Há ao longo do livro um cuidado particular na devida introdução dos inúmeros conceitos que o mesmo analisa e discute. De forma a facilitar a identificação desses conceitos, colocamos na margem esquerda do respectivo texto a marca visual "Conceito" conforme apresentado neste parágrafo. Recomenda-se ao leitor a utilização do índice remissivo para consultar a definição de qualquer dos conceitos tratados neste livro.

Por fim, relativamente à representação de diagramas será utilizada, sempre que for adequado, e por razões óbvias, a linguagem UML.

Agradecimentos

Um agradecimento muito especial à minha família por todo o amor e suporte que tive para poder realizar mais este trabalho, bem como pelas inúmeras horas roubadas ao seu convívio.

Um agradecimento também aos colegas do Judo Clube Portugal e outros amigos cujo convívio me proporcionou os momentos de relaxamento necessário para a produção deste livro.

Parte significativa da actividade que conduziu à realização deste livro foi desenvolvida no âmbito de duas instituições que procuram a excelência - o Departamento de Engenharia Informática do Instituto Superior Técnico e o Instituto de Engenharia de Sistemas e Computadores, às quais não posso deixar de endereçar o meu expresso agradecimento, bem como a todos os colegas e alunos com quem tive o privilégio de conviver, aprender e ensinar durante este período. Em particular, aos alunos da primeira e segunda edição da Pós-Graduação em Sistemas de Informação (POSI'1999 e POSI'2000) do Instituto Superior Técnico, com os quais ensaiei e testei uma parte preliminar deste livro; ao núcleo organizativo do POSI, nomeadamente aos Prof. José Tribolet e Prof. Paulo Guedes, pelo convite que me endereçaram; e ao meu monitor desses cursos, Eng. Miguel Goulão, com quem discuti alguns dos tópicos e exemplos apresentados.

Um agradecimento à editora Centro Atlântico, na pessoa do Dr. Libório Manuel Silva, pelo seu interesse imediato na publicação do livro e pela sua activa e persistente atitude de estar no nosso pequeno mercado nacional de literatura técnico-científica.

Por fim, um agradecimento a todos os colegas que de uma forma ou outra sugeriram, comentaram ou apenas criticaram partes preliminares deste trabalho, ou com quem simplesmente fui partilhando a "ideia" do livro.

Alberto Silva

Quero em primeiro lugar agradecer à minha família, pela sua dedicação, carinho e apoio incondicional, sem a colaboração da qual dificilmente teria participado neste projecto. Quero também agradecer aos meus amigos, de cujo convívio tive que prescindir para poder completar este livro.

Para a realização bem sucedida deste meu projecto foi também decisiva a contribuição de todos os meus colegas da Mentor IT, com os quais tenho abordado alguns temas que são explorados neste livro. A experiência adquirida nos vários projectos em que participei permitiram-me solidificar conhecimentos e sustentar algumas opiniões emitidas neste livro.

Um factor decisivo para a minha participação neste livro foi a experiência como docente, especialmente na Universidade Autónoma de Lisboa, onde tenho estado ligado a disciplinas relacionadas com os temas abordados neste livro. Nesse sentido, gostaria de agradecer ao Prof. José Luís Ferreira e ao Eng. Miguel Gonçalves toda a colaboração e incentivo que me têm dado, bem como o seu contributo em termos de algumas opiniões. Um agradecimento particular a todos os alunos das várias disciplinas que leccionei, pois o esforço de preparação das mêsmas contribuiu para a evolução do conteúdo de uma parte significativa deste livro.

Um agradecimento também para outros colegas com quem mantive, ao longo destes meses de trabalho, uma permuta de opiniões e críticas que me ajudaram a melhorar a qualidade da presente obra.

Finalmente, à Editora Centro Atlântico e ao Dr. Libório Manuel Silva deixo um agradecimento pelo seu interesse na publicação desta obra técnico-científica, valorizando a missão de educar para o futuro.

Carlos Videira

Contactos

Comentários técnicos, sugestões, pedidos de livros ou pedidos de esclarecimentos podem ser dirigidos ao Centro Atlântico (via *www.centroatlantico.pt* ou *geral@centroatl.pt*) que os encaminhará aos autores via correio electrónico se a sua colaboração for necessária.

Autores

Alberto Manuel Rodrigues da Silva é professor auxiliar no Departamento de Engenharia Informática do IST/UTL, investigador sénior no INESC e consultor informático em diferentes empresas e instituições. Tem um doutoramento em Engenharia Informática e Computadores pelo IST/UTL, um mestrado em Engenharia Electrotécnica e Computadores pelo IST/UTL e uma licenciatura em Engenharia Informática pela FCT/UNL. Lecciona actualmente cadeiras da área de Sistemas de Informação e de Engenharia de Software de nível licenciatura, pós-graduação e mestrado. Supervisiona a realização de vários trabalhos finais de curso e de teses de mestrado. Tem interesses profissionais e científicos em sistemas de informação distribuídos em larga escala e em aplicações Web; modelização de software, processos de desenvolvimento de software; e negócios suportados electronicamente. É autor de 2 livros técnicos e cerca de 30 artigos científicos em revistas, conferências e workshops nacionais e internacionais.

Carlos Alberto Escaleira Videira é actualmente *Consulting Manager* na *Mentor*IT, empresa de consultoria estratégica na área dos sistemas de informação, e assistente no Departamento de Ciências e Tecnologias da UAL. Desempenhou funções de coordenação na área de Infor-

mática em diferentes empresas e participou em diversos projectos como consultor. Tem um mestrado em Engenharia Electrotécnica e Computadores pelo IST/UTL e uma licenciatura em Engenharia Informática pela FCT/UNL. Lecciona actualmente cadeiras da área de Planeamento de Sistemas de Informação, Engenharia de Software e Negócios Electrónicos de nível de licenciatura e pós-graduação. Tem interesses profissionais e científicos em temas relacionados com Planeamento Estratégico de Sistemas de Informação, Engenharia de Software, Sistemas de Informação, Gestão de Projectos e Negócios Electrónicos.

Lisboa, Março de 2001

Alberto Manuel Rodrigues da Silva

Carlos Alberto Escaleira Videira

Índice

PARTE 2 – LINGUAGEM DE MODELAÇÃO UML 111

PARTE 3 – METODOLOGIAS DE DESENVOLVIMENTO DE SOFTWARE 289

Parte 1 – Introdução e Visão Geral

Uma empresa de software de sucesso é aquela que consistentemente produz software de qualidade que vai ao encontro das necessidades dos seus utilizadores. Uma empresa que consegue desenvolver tal software, de forma previsível, cumprindo os prazos, com uma gestão de recursos, quer humanos quer materiais, eficiente e eficaz, é uma empresa que tem um negócio sustentado.

Grady Booch, James Rumbaugh,
Ivar Jacobson. *The Unified Modeling Language User Guide*.

Fazer software não é uma tarefa fácil. Fazer software de qualidade é ainda mais difícil. A generalidade dos resultados obtidos ao longo do tempo têm sistematicamente apresentado padrões de baixa qualidade, de custos e prazos completamente ultrapassados. Neste aspecto, a indústria de software deve ser caso único na sociedade actual, pois

apesar da taxa de sucesso dos projectos ser relativamente baixa, o interesse das organizações pelo desenvolvimento de sistemas informáticos tem aumentado constantemente, não se vislumbrando qualquer alternativa. Tudo isto porque as organizações reconhecem que o recurso informação é estratégico e fonte de vantagens competitivas importantes.

O facto dos resultados dos projectos informáticos estarem normalmente abaixo das expectativas e dos diversos problemas que de forma consistente vêm ocorrendo desde o início da utilização das tecnologias de informação, torna extremamente relevantes as várias iniciativas que possam ser desenvolvidas com o objectivo de ultrapassar estes problemas. Sobretudo, vale a pena analisar os diversos esforços que foram efectuados ao longo do tempo, e perceber por que alguns não foram totalmente efectivos na resolução dos problemas, enquanto outros, bem sucedidos, são apontados como melhores práticas a aplicar sistematicamente.

Esta primeira parte do livro pretende dar um enquadramento das questões relacionadas com o desenvolvimento de software, de forma a "aguçar o apetite" dos leitores para os capítulos subsequentes do livro, onde são apresentadas várias ideias, técnicas, métodos e ferramentas que os autores deste livro acreditam que poderão desempenhar um papel decisivo na melhoria dos diversos problemas referidos na primeira parte.

Organização da Parte 1

O Capítulo 1, "Enquadramento e Conceitos Gerais", faz o enquadramento e define o âmbito do livro em questões mais vastas relacionadas com as tecnologias de informação, de forma a transmitir a mensagem ao utilizador que há questões importantes relacionadas com o desenvolvimento de software cuja resolução passa pela realização de actividades e aplicação de técnicas que saem fora do âmbito deste livro. Apresenta ainda os problemas que os sistemas de informação enfrentam actualmente e algumas definições que são relevantes para a compreensão do livro.

O Capítulo 2, “O Processo de Desenvolvimento de Software”, pretende fornecer ao leitor uma visão geral sobre as actividades relacionadas com o desenvolvimento de software, nomeadamente sobre a sua organização, sequência e objectivos a atingir. São ainda clarificados alguns conceitos relacionados com as etapas do desenvolvimento de software.

O Capítulo 3, “Evolução das Metodologias de Desenvolvimento de Software”, procura dar uma visão histórica de como o desenvolvimento de software foi encarado ao longo do tempo, na perspectiva da aplicação de metodologias e respectivas técnicas, e quais as principais motivações para os diversos saltos qualitativos que ocorreram.

Capítulo 1 - ENQUADRAMENTO E CONCEITOS GERAIS

Tópicos

- Introdução
- O Impacto das Tecnologias de Informação
- Produto e Processo
- Sistemas de Informação
- Arquitectura de Sistemas de Informação
- Objectivos do Desenvolvimento de Sistemas de Informação
- Problemas no Desenvolvimento de Sistemas de Informação
- Planeamento Estratégico de Sistemas de Informação
- Engenharia de Software
- Conclusão
- Exercícios

1.1 Introdução

O objectivo deste livro é apresentar a linguagem de modelação UML (Parte 2) e demonstrar a sua aplicação de forma a facilitar todo o desenvolvimento de software, quer seja directamente como técnica de modelação de software, quer seja na sua utilização em metodologias de desenvolvimento (Parte 3) ou em ferramentas de apoio (Parte 4).

De forma a compreender as principais razões por que muitos de nós, ligados à área académica e profissional das tecnologias de informação, acreditamos que o UML representa já actualmente um papel relevante no desenvolvimento de software, é importante enquadrar o leitor deste livro no que consideramos os principais problemas, objectivos e conceitos relacionados com os sistemas de informação e com o seu desenvolvimento. Neste primeiro capítulo, esta abordagem será efectuada de forma ainda muito genérica, e será concretizada nos dois capítulos seguintes.

É ainda importante que o leitor compreenda a relevância de outros conceitos e actividades, que devem ser aplicados no âmbito dos sistemas de informação, mas que não se encontram no âmbito deste livro; estamos a falar, por exemplo, das noções de arquitectura de sistemas de informação e do planeamento estratégico de sistemas de informação. São áreas que estão ao nível da concepção de sistemas de informação, com preocupações de natureza estratégica e que apenas serão brevemente equacionadas neste livro.

1.2 O Impacto das Tecnologias de Informação

É hoje em dia lugar comum ouvir-se falar da importância que a informática ocupa na nossa vida. O impacto e a rápida evolução ao longo dos últimos 40 anos das tecnologias relacionadas com os sistemas de informação tem colocado sucessivos desafios às empresas. De forma a tirar partido das potencialidades destas tecnologias, é necessário um grande investimento em software e hardware. Este impacto é visível não só nas grandes organizações de âmbito internacional, mas atinge também as pequenas e médias empresas.

Desde que surgiram, as tecnologias de informação potenciaram o aparecimento de novas indústrias, como sejam as consultoras de sistemas de informação ou as relacionadas com negócios na Internet, ou reforçaram a importância de outras, nomeadamente as ligadas à indústria de telecomunicações. Têm também provocado uma redefinição das responsabilidades e das interacções entre os parceiros da cadeia de valor de várias indústrias. Nos anos mais recentes, as tecnolo-

gias de informação têm mesmo posto em causa modelos tradicionais de fazer negócio.

Ao longo do tempo, o papel das tecnologias de informação nas organizações sofreu diversas alterações. Actualmente, as tecnologias de informação encontram-se na origem de mudanças significativas ao nível dos modelos de negócio das empresas, e constituem um elemento fundamental para a obtenção de vantagens estratégicas e competitivas. Por isso, a respectiva implementação nas organizações deve ser cuidadosamente planificada e estruturada, de modo a garantir o alinhamento com os objectivos estratégicos do negócio.

A implementação de sistemas de informação requer um investimento significativo (financeiro, tecnológico e de recursos humanos), pelo que estas intervenções deverão merecer o apoio e o comprometimento das administrações. A justificação destes volumes de investimento deve ser efectuada demonstrando qualitativamente e quantitativamente o seu valor estratégico e o impacto positivo nas organizações.

No entanto, muitos gestores não conseguem perceber o verdadeiro alcance de todas estas tecnologias, quer por questões de formação, quer pela sua anterior experiência com sistemas antiquados e obsoletos, que constituíam verdadeiros entraves à satisfação dos requisitos do negócio, e não funcionavam como potenciadores do seu crescimento. Por outro lado, os intervenientes da área de informática criaram no passado uma imagem muito técnica, pouco alinhada com as reais necessidades do negócio, o que contribuiu decisivamente para a não caracterização da informática como uma área estratégica dentro das empresas.

A progressiva importância que os sistemas de informação têm nas organizações pode ser constatada através de diversas situações:

- No passado era comum o responsável da informática depender hierarquicamente do director financeiro, enquanto este reportava directamente à administração. Pelo contrário, actualmente são cada vez menos as organizações em que esta situação se mantém, ficando a área de informática ao mesmo nível que os restantes departamentos e reportando directamente ao órgão que define as respectivas estratégias, a administração; a informática passa assim a ser considerada como uma área estratégica.
- A indústria de software, ou de forma mais geral todas as relacionadas com as tecnologias de informação, é actualmente uma das mais importantes em todo o planeta e uma das principais responsáveis pelo crescimento contínuo da economia mundial durante a última década. Este fenómeno é também visível ao nível das individualidades, já que o homem mais rico do mundo é actualmente um dos principais responsáveis pela maior empresa de software (estamos obviamente a falar de Bill Gates e da Microsoft).
- A crescente importância das empresas relacionadas com a Nova Economia (que de forma simplificada poderemos associar ao fenómeno Internet), cujas acções são transaccionadas nos Estados Unidos num bolsa de valores específica (Nasdaq).
- A importância destas empresas tem motivado a crescente preocupação dos governos em garantir o acesso livre ao mercado e a tentar evitar posições monopolistas. É o caso do presente litígio entre o governo americano e a Microsoft, onde assistimos à disputa em torno de questões por vezes pouco racionais; no entanto, e independentemente da nossa posição pessoal, o governo americano actua de forma semelhante à dos seus antecessores há algumas décadas atrás, em relação a empresas de outras indústrias chave, como eram na altura a do petróleo e do aço.

Muitos outros exemplos poderiam ser dados, mas a conclusão óbvia é que nos tornámos dependentes das tecnologias de informação, quer do ponto de vista pessoal quer profissional.

1.3 Produto e Processo

A importância das tecnologias de informação na nossa vida é sobretudo concretizada pelas funcionalidades que são implementadas ao nível do software, e que são disponibilizadas com o suporte de um conjunto de dispositivos diversos (hardware). O primeiro pode ser considerado o componente lógico dos sistemas de informação, o segundo o componente físico.

Não existe uma definição rigorosa e inequívoca de software. Diversos autores [Pressman2000, Schach1999] encaram o software como o resultado final de um processo, ao qual designam por “Engenharia de Software”. O que é um facto é que o software não é dádiva da natureza, nem é objecto de uma produção numa linha de montagem, realizada de forma perfeitamente automática, sem qualquer intervenção humana, criativa e subjectiva.

Quando falamos em "processo" esta palavra implica desde logo a definição de um conjunto de actividades uniformizadas, a aplicar sistematicamente, que se encontram agrupadas em fases. Cada uma destas fases tem os seus intervenientes, aos quais são atribuídas responsabilidades, que possui diversos *inputs* e que produz *outputs*. Do ponto de vista da garantia da qualidade do produto final (o software), é fundamental que o processo seja realizado segundo parâmetros que permitam também aferir a respectiva qualidade, isto é, não conseguiremos optimizar o resultado final sem uma preocupação no processo que o produz.

Se pensarmos que o desenvolvimento do software é um processo que deve ser baseado na aplicação de técnicas e práticas rigorosas, sistemáticas, eficientes e controláveis, podemos concluir que este se aproxima bastante de outras realizações humanas, como a construção de qualquer obra de engenharia civil (por exemplo, a construção da ponte Vasco da Gama em Lisboa). Daí o nome de "Engenharia de Software" precisamente como tentativa de trazer para esta actividade a preocupação da aplicação de técnicas de engenharia ao desenvolvimento de software, por exemplo, modelar antes de realizar; estimar

diversos factores antes de avançar; medir antes, durante e depois do produto realizado; analisar factores de risco.

Para além dos elementos já descritos, tal como nas outras engenharias, também a realização efectiva das funções de desenvolvimento de software pressupõe a utilização de ferramentas de apoio a todo o processo. O tempo em que o desenvolvimento era efectuado de forma completamente manual já não é razoável actualmente (tal como ninguém constrói uma casa, e muito menos uma ponte, unicamente à custa do seu esforço físico). As características destas ferramentas podem ter um impacto apreciável no produto final (bem como no processo), e a demonstração desse facto é um dos objectivos deste livro.

No entanto, é também importante esclarecer desde já que a produção de software encerra em si mesma alguma subjectividade, devido ao facto de ser realizada por seres humanos, que em diversos pontos podem introduzir factores resultantes da opinião pessoal (e que até certo ponto podem ser benéficos, pois a criatividade pode levar à produção de software com melhor aceitação e desempenho). Neste aspecto, o processo aproxima-se mais de uma actividade artística do que propriamente uma actividade de engenharia. É por isso que nós consideramos, tal como outros autores, que o ponto de equilíbrio correcto depende de cada caso, mas deve-se encontrar a meio caminho entre a aplicação de técnicas estruturadas (Engenharia) e introdução de factores de criatividade (Arte).

Actualmente, e num contexto social e económico em constante mudança, espera-se que o software seja capaz de evoluir a um ritmo que não ponha em causa o crescimento das organizações. São por isso fundamentais as seguintes características:

- Flexibilidade, enquanto capacidade de evolução face aos requisitos de negócio.
- Fiabilidade, o que implica que o número de problemas ocorrido seja reduzido e não ponha em causa o funcionamento das organizações.
- Implementação das necessidades das organizações
- Nível de desempenho adequado

- Facilidade de utilização, com uma interface amigável e intuitiva para o utilizador.

1.4 Sistemas de Informação

A visão mais tradicional sobre o conceito de **software** limita-se a considerá-lo como um conjunto de programas, constituído por blocos de código. Outros autores englobam ainda neste conceito a documentação de apoio que é produzida. No entanto, quando falamos actualmente do componente lógico que serve de suporte às necessidades das organizações, o conceito mais abrangente normalmente utilizado é o de sistemas de informação.

Tal como em muitas outras situações no domínio da informática, não existe uma definição formal e consensual deste conceito. Neste livro adoptaremos a seguinte definição: um **sistema de informação** é um conjunto integrado de recursos (humanos e tecnológicos) cujo objectivo é satisfazer adequadamente a totalidade das necessidades de informação de uma organização e os respectivos processos de negócio. Nesta definição o conceito **processo de negócio** pretende representar uma sequência de actividades, que processam vários *inputs* e produzem vários *outputs* e que possuem objectivos. Pode ser realizado por pessoas e/ou de forma automática. Exemplos de processos de negócio incluem as compras de matérias-primas, a contratação de um empregado ou a distribuição de produtos acabados.

Existem outras definições para o conceito de sistema de informação que enumeram os respectivos componentes, nomeadamente pessoas, hardware, software, redes e dados, sempre numa perspectiva integrada, e de modo a suportar e melhorar as operações diárias do negócio, bem como a satisfazer as necessidades de informação dos gestores [O'Brien00]. Finalmente, de referir que alguns autores não consideram a parte de processos manuais como fazendo parte do sistema de informação.

Os sistemas de informação são actualmente considerados essenciais para suportar adequadamente estratégias de globalização e de reengenharia de processos de negócio e para a obtenção de vantagens

competitivas, com impacto ao nível da redução de custos, estratégias de diferenciação e/ou de inovação, promovendo e facilitando as relações e negócio com parceiros e clientes. É objectivo fundamental dos sistemas de informação garantir o alinhamento das tecnologias da informação com os objectivos estratégicos do negócio.

O impacto dos sistemas de informação nas organizações é inegável e inevitável. Uma das mais antigas classificações de sistemas de informação foi proposta por Anthony em 1965 [Anthony65]. Esta classificação agrupava os sistemas de informação em função do nível das actividades de gestão dentro da organização no qual o software tem impacto:

- Operacional, onde se incluíam todos os sistemas de informação que suportavam directamente as operações do dia-a-dia. Estamos a falar sobretudo de operações que implicam alterações na informação.
- Táctico, que inclui as funcionalidades de análise de informação, sobretudo orientadas para suportar o processo de tomada de decisões com impacto na gestão de curto prazo.
- Estratégico, essencialmente preocupado com questões de planeamento, em que o impacto se situa temporalmente no médio e longo prazo.

Tipo de Sistemas	**Exemplos**
Operacionais	Facturação, Controlo de encomendas, Contabilidade geral, Controle de Stocks, Salários
Tácticos	Análise de vendas, Controle orçamental, Contabilidade analítica, Gestão do inventário, Análise da qualidade
Estratégicos	Previsão de vendas, Planeamento da alocação da produção, Planeamento recursos humanos, Previsão de receitas e custos, Modelação financeira

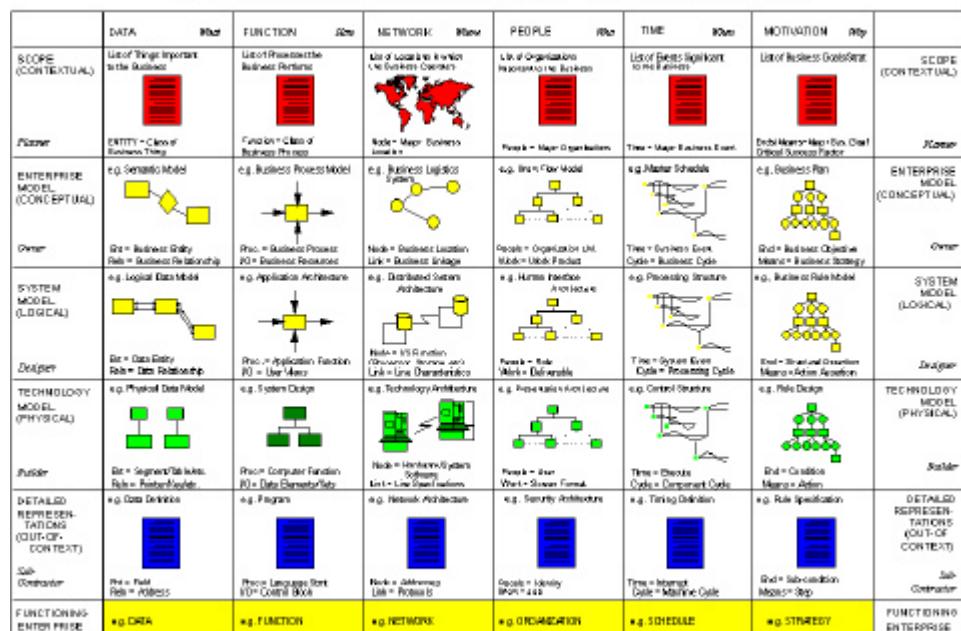
Tabela 1.1: Exemplos de sistemas de informação segundo a classificação de Anthony.

Muitas outras classificações existem, segundo parâmetros variados, mas a sua apresentação sai fora do âmbito deste livro.

1.5 Arquitectura de Sistemas de Informação

A crescente complexidade dos sistemas de informação e a dificuldade de apresentação da sua estrutura aos diversos interessados, incluindo utilizadores e informáticos, motivou durante a década de 80 e inícios da década de 90 um conjunto de esforços no sentido de formalizar e uniformizar a respectiva apresentação, de modo a garantir, adicionalmente, a integração dos diversos componentes de informação da organização.

Em 1987, John Zachman publicou o artigo "*A Framework for Information Systems Architecture*" [Zachman87], em que introduzia o conceito de arquitectura de sistemas de informação. As ideias propostas resultaram de conhecimentos e experiências de outras disciplinas mais antigas (arquitectura, engenharia da produção) e rapidamente se tornaram numa referência para todos aqueles que têm algum interesse no tema da arquitectura de sistemas de informação. Infelizmente, e apesar da relevância do tema, muitos destes conceitos continuam desconhecidos da maioria do público informático.

De acordo com este autor, a **arquitectura** é o “conjunto de representações descritivas (modelos) relevantes para a descrição de um objecto, de forma a que este possa ser construído de acordo com os requisitos (de qualidade) e mantido ao longo da sua vida útil”.

Esta definição é consideravelmente genérica e informal e não indica o âmbito do termo arquitectura; de facto, no caso da abordagem proposta por Zachman, ela refere-se quer aos sistemas de informação quer à empresa, uma vez que o mesmo modelo apresenta relativamente a cada conceito a perspectiva do negócio e dos sistemas de informação.

O *Framework de Zachman* é uma estrutura lógica de classificação e apresentação dos modelos de uma organização relevantes para a respectiva gestão, bem como para o desenvolvimento dos seus sistemas, e pode ser observado na Figura 1.1. Nesta perspectiva, modelar um sistema significa determinar e representar um conjunto de informação, sobre

vários tópicos (colunas da matriz), relevante para vários intervenientes (linhas da matriz).

ENTERPRISE ARCHITECTURE - A FRAMEWORK ™

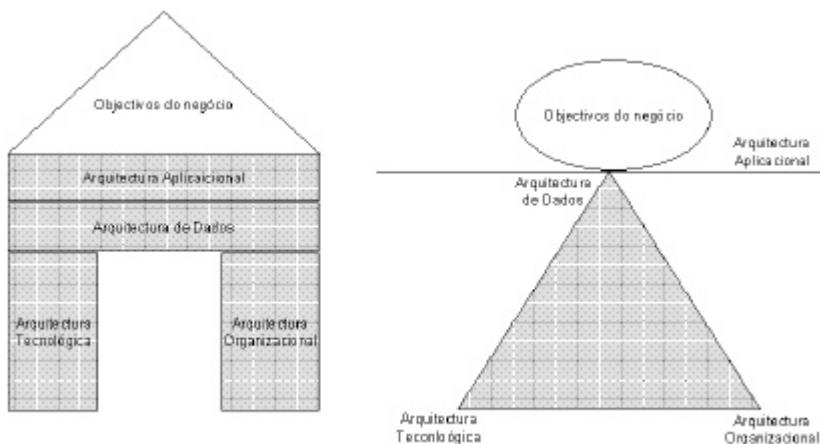
	DATA *What*	FUNCTION *How*	NETWORK *Where*	PEOPLE *Who*	TIME *When*	MOTIVATION *Why*	
SCOPE (CONTEXTUAL) *Planner*	List of Things Important to the Business ENTITY = Class of Business Thing	List of Processes the Business Performs Function = Class of Business Process	List of Locations in which the Business Operates Node = Major Business Location	List of Organizations Important to the Business People = Major Organizations	List of Events Significant to the Business Time = Major Business Event	List of Business Goals/Strat. Ends/Means = Major Bus. Goal/ Critical Success Factor	SCOPE (CONTEXTUAL) *Planner*
ENTERPRISE MODEL (CONCEPTUAL) *Owner*	e.g. Semantic Model Ent = Business Entity Rel = Business Relationship	e.g. Business Process Model Proc. = Business Process I/O = Business Resources	e.g. Business Logistics System Node = Business Location Link = Business Linkage	e.g. Work Flow Model People = Organization Unit Work = Work Product	e.g. Master Schedule Time = Business Event Cycle = Business Cycle	e.g. Business Plan End = Business Objective Means = Business Strategy	ENTERPRISE MODEL (CONCEPTUAL) *Owner*
SYSTEM MODEL (LOGICAL) *Designer*	e.g. Logical Data Model Ent = Data Entity Rel = Data Relationship	e.g. Application Architecture Proc. = Application Function I/O = User Views	e.g. Distributed System Architecture Node = I/S Function [illegible] Link = Line Characteristics	e.g. Human Interface Architecture People = Role Work = Deliverable	e.g. Processing Structure Time = System Event Cycle = Processing Cycle	e.g. Business Rule Model End = Structural Assertion Means = Action Assertion	SYSTEM MODEL (LOGICAL) *Designer*
TECHNOLOGY MODEL (PHYSICAL) *Builder*	e.g. Physical Data Model Ent = Segment/Table/etc. Rel = Pointer/Key/etc.	e.g. System Design Proc.= Computer Function I/O = Data Elements/Sets	e.g. Technology Architecture Node = Hardware/System Software Link = Line Specifications	e.g. Presentation Architecture People = User Work = Screen Format	e.g. Control Structure Time = Execute Cycle = Component Cycle	e.g. Rule Design End = Condition Means = Action	TECHNOLOGY MODEL (PHYSICAL) *Builder*
DETAILED REPRESEN-TATIONS (OUT-OF-CONTEXT) *Sub-Contractor*	e.g. Data Definition Ent = Field Rel = Address	e.g. Program Proc.= Language Stmt I/O = Control Block	e.g. Network Architecture Node = Addresses Link = Protocols	e.g. Security Architecture People = Identity Work = [illegible]	e.g. Timing Definition Time = Interrupt Cycle = Machine Cycle	e.g. Rule Specification End = Sub-condition Means = Step	DETAILED REPRESEN-TATIONS (OUT-OF-CONTEXT) *Sub-Contractor*
FUNCTIONING ENTERPRISE	e.g. DATA	e.g. FUNCTION	e.g. NETWORK	e.g. ORGANIZATION	e.g. SCHEDULE	e.g. STRATEGY	FUNCTIONING ENTERPRISE

Figura 1.1: Framework de Zachman.

Este diagrama apresenta a relação entre as diferentes funções que podem ser identificadas na organização, e a visão e detalhe que têm (e precisam de ter) sobre os diversos objectos e conceitos da organização. Assim, são considerados cinco perfis de intervenientes que se relacionam com o sistema:

- *Planner*, responsável pelo planeamento estratégico da organização.
- *Owner*, responsável pela operação do negócio.
- *Designer*, responsável pela elaboração da especificação funcional do sistema.
- *Builder*, responsável pela elaboração da especificação técnica do sistema.
- *Sub-contractor*, responsável pela especificação detalhada e construção do sistema.

Os dois primeiros níveis são tipicamente utilizadores do sistema e relacionados com as áreas do negócio, enquanto os três últimos são intervenientes com perfil informático. À medida que se desce na hierarquia, aumenta o nível de detalhe a que a análise e a modelação têm que ser efectuadas. Cada um destes perfis tem uma visão diferente sobre um conjunto de factores analisados pelo *framework*, designadamente:

- Qual a constituição do sistema (*What*) - os dados?
- Como é que o sistema funciona (*How*) – as funções?
- Onde está localizado o sistema (*Where*) – as relações e as redes?
- Quem são os interessados no sistema (*Who*) – as pessoas?
- Quando ocorrem factos relevantes no sistema (*When*) – o tempo?
- Porque é que o sistema funciona assim (*Why*) – as motivações?

Este tipo de abordagem muito estruturada permite utilizar um único modelo para simplificar a compreensão e comunicação sobre a visão da organização; dar ênfase à análise de variáveis independentes; e manter uma perspectiva disciplinada sobre relações necessárias para preservar a integridade dos conceitos da organização. Pode ser utilizada nas diferentes fases do ciclo de desenvolvimento de sistemas de informação, desde o planeamento estratégico até ao desenho técnico detalhado.

Uma outra abordagem alternativa baseia-se no *Framework de Index* [Wurman97], e considera que a arquitectura de sistemas de informação é um conjunto integrado e consistente de componentes, que são definidos de forma a garantir o respectivo alinhamento com os objectivos de negócio, e por isso são suportados por todos os elementos da organização. Estes componentes encontram-se normalmente organizados em quatro grandes blocos:

- Arquitectura aplicacional: conjunto de sistemas e aplicações necessários para suportar os objectivos de negócio da organização.
- Arquitectura tecnológica: componentes de infra-estrutura e máquinas necessários para suportar as funcionalidades e requisitos das aplicações identificadas.
- Arquitectura de dados: conceitos e entidades necessárias à execução dos processos de negócio da organização.

- Arquitectura organizacional: estrutura de recursos humanos necessária para suportar adequadamente os restantes componentes dos sistemas de informação.

A definição destes componentes deve obedecer a uma sequência lógica, que tem a ver com as precedências e as interligações que existem entre eles. Como o componente que suporta os objectivos de negócio são as aplicações, estas devem ser identificadas em primeiro lugar, em paralelo com os conceitos (dados) que gerem. As componentes tecnológica e organizacional serão as últimas a ser definidas, de forma a suportarem adequadamente as restantes.

Figura 1.2: Representações da Arquitectura de Sistemas de Informação.

A Figura 1.2 ilustra de uma forma esquemática e simbólica a importância da definição de uma arquitectura estável em que os diversos componentes estão relacionados entre si de forma equilibrada. A parte esquerda da Figura 1.2 pretende representar uma arquitectura estável, em que os componentes estão solidamente integrados, ao contrário do que acontece na parte direita da mesma figura, em que a arquitectura é claramente instável e o seu equilíbrio deficiente.

1.6 Objectivos do Desenvolvimento de Sistemas de Informação

Em 1983, Robert Block definiu um sistema de informação bem sucedido como sendo aquele que é produzido dentro do prazo e nos custos estimados; é fiável (sem erros e disponível quando necessário) e pode ser mantido facilmente e a baixo custo; responde adequadamente aos requisitos definidos; e satisfaz os utilizadores. Esta definição, demasiado restrita, leva à conclusão natural de que poucos serão os sistemas que respeitam estes requisitos [Block83].

Ao longo do tempo, o papel do software e dos sistemas de informação nas organizações tem evoluído de forma a posicionar-se cada vez mais como factor estratégico e competitivo. Nos primórdios da computação (há apenas 50 anos atrás), o software era utilizado sobretudo para a resolução de problemas de cálculo relacionados com questões militares (por exemplo, cálculo das trajectórias de projecteis). Os primeiros computadores com aplicações de natureza comercial eram utilizados pelas grandes organizações com o objectivo de automatizar algumas das etapas dos processos de negócio e desta forma reduzir custos. A partir deste momento a importância e impacto dos sistemas de infor-mação nas organizações não tem parado de crescer, e podemos carac-terizá-la resumidamente de acordo com o apresentado na Figura 1.3.

Figura 1.3: Factos relevantes na evolução dos sistemas de informação.

Outras classificações foram elaboradas, nomeadamente a de Primozic [Primozic90], que identifica cinco grandes ondas de inovação, de acordo com a evolução das tecnologias de informação e os benefícios crescentes que oferecem às organizações.

Onda de Inovação	**Utilização Funcional**	**Impacto na Organização**
Reduzir Custos	Administrativas	Gestão de processos
Potenciar Investimentos	Financeiras, Produção	Gestão de recursos
Melhorar e aumentar produtos e serviços	Marketing, Distribuição, Apoio ao Cliente	Crescimento e aumento da quota de mercado
Melhorar a eficácia das decisões	Decisões Estratégicas	Reengenharia da organização
Atingir o consumidor	Funcionalidades nos computadores dos clientes	Reestruturação da indústria

Tabela 1.2: Ondas de Inovação de Primozic.

Independentemente destas classificações, existe um conjunto de razões que levam as organizações a investir em sistemas de informação e que podemos indicar de seguida, de forma resumida:

- Reduzir custos operacionais, através da automatização e reformulação dos processos de negócio.
- Satisfazer requisitos de informação dos utilizadores.
- Contribuir para a criação de novos produtos e serviços.
- Melhorar o nível de serviço prestado aos clientes actuais e facilitar a aquisição de novos clientes.
- Melhorar e automatizar a relação com os parceiros de negócio.
- Melhorar o desempenho de pessoas e máquinas.

1.7 Problemas no Desenvolvimento de Sistemas de Informação

Historicamente, o software tem apresentado de forma sistemática e contínua os mesmos problemas. As razões que no passado justificaram a adopção de métodos de trabalho mais estruturados continuam a verificar-se e por isso somos levados a concluir que estas iniciativas não vieram, afinal, resolver de todo os problemas. Se pensarmos no impacto na organização, estes podem ser essencialmente agrupados em três níveis:

- Falta de qualidade, traduzida na satisfação incompleta dos requisitos e nos problemas que se verificam após a instalação do produto.
- Desvios dos prazos previamente estabelecidos para o desenvolvimento de software.
- Custos previamente definidos para o desenvolvimento de software largamente ultrapassados.

A Tabela 1.3 ilustra como ao longo do tempo os diversos problemas têm existido de forma contínua, e independentemente das iniciativas que têm surgido, estas não eliminaram de forma alguma o problema.

1979	Em 57 projectos, 46% estavam atrasados (média de 7 meses) e 59% encontravam-se acima do orçamento [Lehman79].
1979	Em 9 projectos, um valor de investimento de $3.2M USD nunca foi completado, $2M USD nunca foi utilizado, $1.3M USD foi abandonado, $0.2M USD foram utilizados com algumas alterações e apenas $0.1M USD foi utilizado como entregue [General79].
1982	75% dos sistemas desenvolvidos nunca foram completados ou utilizados [Gladden72].
1984	Em 2000 empresas, 40% dos sistemas falharam o atingimento dos resultados esperados [Bikson84].
1987	75% dos sistemas de controle da produção e inventário implementados tiveram problemas [Works87].

1988	Num universo de 34 analistas de sistemas, 70% consideraram que entre 20% e 50% dos projectos falham, porque não são satisfeitos os requisitos de negócio previstos [Lyytinen].
1994	Em 82 executivos entrevistados, 22% tinham abandonado mais de 5 projectos nos 5 anos anteriores e 69% abandonaram pelo menos 1 [Ewusi-Mensah].
1995	Em 143 projectos, 25% não respondiam aos requisitos [Phan95].
1995	Num universo de 365 empresas, 31% projectos cancelados antes do fim, 53% ultrapassaram custos; só 12% de 3682 foram completados a tempo e nos custos previstos [Johnson95].

Tabela 1.3: Estatísticas diversas obtidas ao longo do tempo sobre os projectos de desenvolvimento de software.

Foi precisamente este tipo de problemas que motivou a designação de "crise no software" já durante a década de 70, a qual foi reforçada por Fred Brooks no seu célebre artigo "*No Silver Bullet*" ("não existem balas de prata") [Brooks86], no qual este refere que dificilmente se encontraria uma cura milagrosa que pudesse resolver os problemas associados ao processo de desenvolvimento de software.

Os problemas até agora referidos têm muito a ver com questões que se verificam durante o processo de desenvolvimento de software, mas igualmente graves são as situações que podem ocorrer depois deste processo estar concluído, e os sistemas entrarem em produção. Neste caso, o adequado funcionamento dos sistemas é crucial para a existência e sobrevivência das organizações e das pessoas envolvidas, a diferentes níveis, envolvendo questões económicas, de segurança, privacidade, qualidade de vida, etc. Existem diversos casos clássicos que apontam para as falhas do software em funcionamento:

- Em 1979, ainda durante o período da guerra-fria, o mundo pode ter estado à beira de uma guerra nuclear quando o sistema americano que controlava o espaço aéreo detectou o lançamento de mísseis pela União Soviética em direcção aos Estados Unidos; de facto, tratava-se de um ataque simulado, e apesar de não terem sido divulgados muitos detalhes, parece legítimo supor que tal se tratou de um erro do sistema [Neumann80].

- Durante a guerra do Golfo, uma falha no software dos mísseis Patriot que os Estados Unidos enviaram para a zona da guerra não foi atempadamente detectada, e a correcção só chegou um dia após um ataque iraquiano com mísseis ter causado a morte a cerca de trinta soldados americanos [Mellor94].
- Devido a um erro no software de controlo de um equipamento médico, pelo menos dois doentes morreram entre 1985 e 1987 em consequência de terem recebido doses exageradas de radiação [Leveson93].
- Problemas diversos no software de controlo da distribuição e encaminhamento de bagagem do aeroporto de Denver, nos Estados Unidos, provocaram custos superiores a 1 milhão USD por dia [Gibbs94].
- Em 1995, estimativas diversas apontavam para que o custo dos projectos de software que foram abandonados nos Estados Unidos equivaleria a cerca de 80 mil milhões USD, qualquer coisa como 1% do PIB americano [Johnson95].

Como se pode constatar dos exemplos anteriores, os problemas resultantes do mau funcionamento ou do processo de desenvolvimento de software podem ter impacto em duas áreas críticas: questões financeiras por um lado, e vidas humanas por outro. Mesmo após a entrada em funcionamento do software, poder-se-ia pensar que o número de problemas iria diminuir drasticamente e estabilizar num nível muito baixo, idealmente próximo de 0. Tal não acontece, o que tem muito a ver com o facto de qualquer intervenção posterior à implementação do software poder vir a gerar um conjunto de problemas não previstos, e consequentemente um acréscimo de erros.

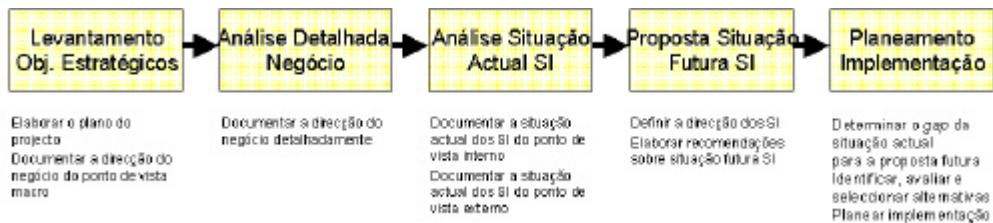
Diversas causas estão na origem deste crónico falhanço dos projectos de sistemas de informação, nomeadamente:

- Falta de empenhamento dos órgãos de topo das organizações.
- Falta de comprometimento e empenhamento dos utilizadores.
- Incompreensão do valor dos sistemas de informação.
- Falta de entendimento e de sintonia entre informáticos e clientes utilizadores do sistema, no âmbito e requisitos do mesmo.
- Deficiências várias no processo de desenvolvimento.

- Falhas na coordenação do projecto, nomeadamente ao nível da definição dos objectivos e das prioridades e da elaboração de estimativas.
- Falta de qualidade e inadequação dos recursos envolvidos.
- Mudanças frequentes dos requisitos do negócio e incapacidade de lidar com esta situação.
- Dificuldades na integração de componentes.
- Qualidade e desempenho do software deficiente, muito relacionados com problemas ao nível do controle de qualidade.
- Incapacidade de identificar e controlar os riscos do projecto.

1.8 Planeamento Estratégico de Sistemas de Informação

No passado, os sistemas de informação foram desenvolvidos simplesmente para melhorar a eficiência de determinadas funções de negócio; mais recentemente passaram a concentrar-se na obtenção de vantagens competitivas. Este facto justifica a inclusão de considerações sobre as tecnologias de informação na definição de estratégias do negócio, tal como é defendido por McFarlan [McFarlan83].

Um dos objectivos dos sistemas de informação é a satisfação adequada dos requisitos de negócio, garantindo assim o correcto alinhamento com a estratégia da organização. É por isso importante que, antes de se iniciar qualquer processo de desenvolvimento de componentes da arquitectura de sistemas de informação, que a mesma seja pensada de um ponto de vista global, garantindo assim a completa integração entre os componentes e a prioritização da respectiva implementação.

É esse o âmbito do Planeamento Estratégico de Sistemas de Informação, cujo principal resultado é um Plano Estratégico de Sistemas de Informação (ou Plano Director de Sistemas), que define os componentes do sistema de informação a implementar, e funciona como um guia para todas as futuras intervenções na área de informática. Na sequência deste plano, devem ser identificadas e prioritizadas as acções a desencadear para atingir a situação futura proposta. Só depois se entra no âmbito do desenvolvimento dos sistemas de informação, naquilo que normalmente se designa por Engenharia de Software. Por

esta razão, é também frequente que a actividade de planeamento estratégico seja designada por Engenharia de Sistemas, para traduzir a ideia de uma perspectiva mais abrangente.

Este tipo de abordagem pode ser aplicado numa organização já existente, sendo nesse caso necessário identificar o diferencial entre a situação actual dos sistemas de informação e o conjunto de recomendações elaborado.

Podemos definir o **Planeamento Estratégico de Sistemas de Informação** (PESI) como um processo cuja finalidade é garantir o alinhamento dos sistemas de informação com os objectivos do negócio ou como Lederer referiu [Lederer88] "o PESI é o processo de decidir os objectivos para a organização informática e identificar as aplicações informáticas potenciais que a organização deve implementar".

Enquanto processo tem uma sequência de fases, cada uma com actividades e objectivos bem identificados (ver Figura 1.4).

Figura 1.4: Metodologia de Planeamento Estratégico de Sistemas de Informação.

O objectivo deste processo é realizar um conjunto de actividades de levantamento de informação, do negócio e dos sistemas de informação, durante as três primeiras fases, de modo a que na quarta fase de possam elaborar recomendações sustentadas e que possibilita a elaboração de planos do projecto. No final do processo de PESI dispõe-se de um plano estratégico de sistemas de informação bem documentado, uma compreensão detalhada da situação actual do negócio e dos sistemas de informação e uma definição da direcção dos sistemas de informação suportada por toda a organização.

1.9 Engenharia de Software

Depois de definida uma estratégia global e identificados os componentes que é necessário desenvolver, a sua concretização passa para o domínio de outra "ciência", a Engenharia de Software. Esta inclui todas as actividades que vão desde um planeamento inicial do projecto até à instalação do sistema em produção, e posterior suporte. Por isso, disciplinas como análise de sistemas, gestão de projectos, programação, controle de qualidade poderão ser incluídas no âmbito da Engenharia de Software, conforme aqui a entendemos.

Uma das primeiras definições de Engenharia do Software foi dada por Fritz Bauer, nos finais da década de 60, como sendo "a definição e utilização de princípios de engenharia sólidos, de modo a desenvolver software económico, fiável e que trabalha eficientemente em máquinas reais. Inclui pois um conjunto de métodos, de ferramentas e de procedimentos". No entanto, esta definição peca por não fazer qualquer referência a aspectos técnicos, não referir a importância da satisfação do cliente, do cumprimento de prazos, da utilização de métricas e não enfatizar a importância de se utilizar um processo maduro.

Uma das definições mais utilizada hoje em dia foi proposta pelo IEEE em 1993, e defende que "a **Engenharia de Software** é a aplicação de um processo sistemático, disciplinado, e quantificado ao desenvolvimento, operação e manutenção de software; ou seja, a **Engenharia de Software** é a aplicação de técnicas de engenharia ao software".

As actividades associadas à Engenharia de Software podem ser agrupadas em três grandes fases, tendo em conta que o seu objectivo é o desenvolvimento e operação de um produto: concepção, implementação e manutenção. Cada uma destas fases pode ainda ser dividida em outras mais elementares (ver Capítulo 2 para mais detalhes). Ao longo de cada fase existem tarefas, subprodutos a desenvolver, pontos de verificação e intervenientes. Existe também um conjunto de actividades de suporte contínuas: gestão de projecto, controle de qualidade, gestão da configuração, elaboração de documentação, elaboração de estimativas, gestão do risco, entre outras. É pois uma área de conhecimento

muito vasta, o que torna ainda mais difícil a sua aplicação de forma rigorosa e sistemática.

Apesar de se reconhecer o valor de um planeamento global com uma perspectiva integradora, muitas organizações estão mais preocupadas em resolver problemas imediatos e parciais, e por isso desencadeiam projectos individuais, que muitas vezes contemplam apenas as necessidades e requisitos de uma área restrita da organização. Assim, o esforço da implementação de sistemas de informação começa logo pelas actividades que já se situam no domínio da engenharia de software.

A Figura 1.5 ilustra de uma forma esquemática a discussão anterior entre as grandes áreas de Planeamento Estratégico de Sistemas de Informação e de Engenharia de Software, e suas relações. Através dela podemos perceber que, por exemplo, a Engenharia de Software inclui diversas questões que pertencem à Gestão de Projectos (planeamento, execução e aompanhamento de um projecto), mas está fora do seu âmbito questões como gestão de recursos humanos. A figura ilustra complementarmente o contexto e o foco primordial deste livro, designadamente os assuntos conotados com as abordagens orientadas por objectos, e em particular o UML.

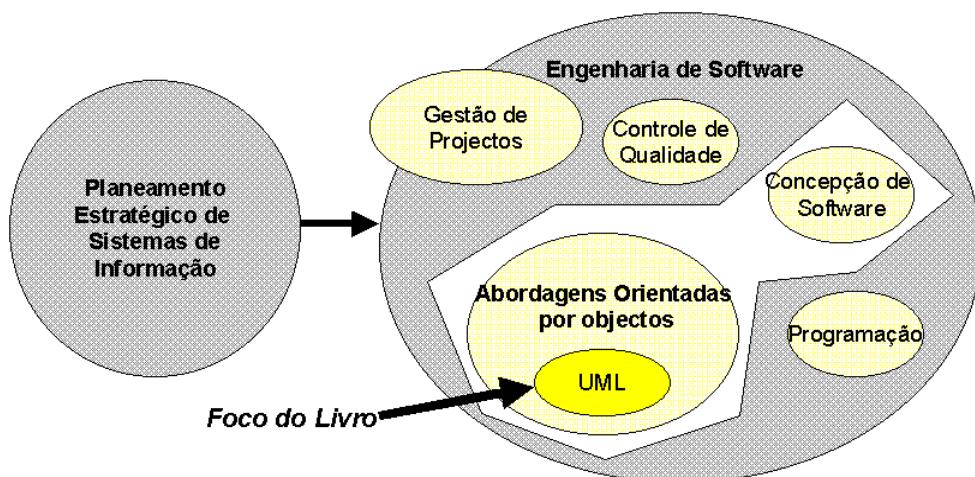



Figura 1.5: Relação entre PESI e Engenharia de Software, e o foco do livro.

Existem diversos produtos construídos pelo homem que apresentam problemas, mas mais raramente que os que se verificam no software. Por exemplo, um frigorífico pode falhar, mas menos do que um programa de contabilidade; uma ponte pode cair, mas tal é com certeza menos frequente do que a ocorrência de problemas nos sistemas operativos dos computadores. A ideia de que a concepção, implementação e manutenção do software poderiam ser realizadas aplicando técnicas tradicionais de engenharia levou a que já em 1967 um grupo de trabalho da NATO propusesse pela primeira vez o termo Engenharia de Software, e que este assunto fosse discutido em larga escala durante a conferência *NATO Software Engineering Conference* realizada na 1968.

A conclusão que resultou desta reunião foi que se deveriam utilizar os princípios e paradigmas de outras disciplinas de engenharia já bem estabelecidas de modo a resolver o que na altura se designou por crise do software (a qualidade do software era inaceitavelmente baixa e os custos e prazos não eram cumpridos). Actualmente há quem considere que a expressão mais adequada não seria “crise” mas sim “depressão”, dada a duração que ela já tem e ao facto de não se vislumbrar uma solução imediata.

1.10 Conclusão

Este capítulo de introdução tem como objectivo dar uma ideia dos principais problemas e preocupações que de um ponto de vista genérico se colocam aos intervenientes no processo de gestão e desenvolvimento informático. Estes problemas têm sido uma constante desde o início do desenvolvimento de software, e independentemente das iniciativas desenvolvidas, não foi possível até à data eliminá-los integralmente.

Devido a esta falta de resultados, poderíamos considerar que estávamos perante um facto consumado e inevitável, e abandonar os esforços no sentido de corrigir as falhas detectadas. A mensagem dos restantes capítulos, e do livro no seu todo, é a de que a nossa postura não pode nem deve ser esta, e que devemos continuar a procurar ultrapassar os problemas existentes, tal como os cientistas em medicina não desistem de procurar a cura para o cancro, apesar de já o tentarem há muitos

anos. Devemos recolher os exemplos das histórias de sucesso e daquilo que se consideram as melhores práticas de desenvolvimento de software. Nos próximos dois capítulos iremos abordar estas questões de um ponto de vista evolutivo e mais abrangente.

Tivemos também como objectivo a introdução e enquadramento sucinto de alguns conceitos desta área de engenharia, nomeadamente os com-ceitos de sistemas de informação, arquitecturas de sistemas de informação, planeamento estratégico de sistemas de informação, e engenharia de software. Esta discussão de conceitos permitiu ainda definir e explicitar claramente o âmbito e o contexto deste livro.

1.11 Exercícios

Ex.1. Explique, com base na definição de Anthony, a diferença entre um sistema de informação operacional, táctico e estratégico. Dê exemplos para clarificar a sua justificação.

Ex.2. Discuta a noção de sistema de informação face à noção de software. Com base na definição dada no livro pode-se ter um sistema de informação sem software? Justifique.

Ex.3. Enumere os três principais problemas relacionados com o desenvolvimento de sistemas de informação. Enumere três das causas conhecidas.

Ex.4. Justifique a importância do PESI (planeamento estratégico de sistemas de informação).

Ex.5. A flexibilidade é frequentemente referida como um dos atributos relacionados com a existência de qualidade num sistema de informação. Discuta os aspectos positivos e identifique possíveis consequências negativas.

Ex.6. De um modo geral, as opiniões expressas pelos executivos do negócio sobre os informáticos são criticas face ao seu conhecimento do negócio, mas não tecem grandes considerações relativamente a questões de natureza técnica. Indique algumas razões por que tal acontece.

Ex.7. Na sua opinião, existem situações em que faz sentido ter um processo conjunto de actividades de planeamento estratégico de sistemas de informação e de engenharia de software? Justifique a sua resposta.

Capítulo 2 - O Processo de Desenvolvimento de Software

Tópicos

- Introdução
- Processos e Metodologias
- Modelos e Modelação
- Boas Práticas no Desenvolvimento de Software
- Fases do Processo de Desenvolvimento de Software
- Processos de Desenvolvimento de Software
- Conclusão
- Exercícios
-

2.1 Introdução

A produção de software é frequentemente uma actividade não estruturada, por vezes caótica, sem orientações de natureza estratégica e sem planos de gestão e controle. Assim se justifica a denominação *build and fix* (programar e corrigir). Como vimos no primeiro capítulo, os problemas associados ao desenvolvimento de software são de tal dimensão que é fundamental a definição e aplicação de princípios, regras e estratégias que conduzam a melhorias significativas em todo o desenvolvimento de software. Estes princípios deverão nortear sempre a intervenção do informático na implementação de sistemas de informação.

Em qualquer desenvolvimento de sistemas de informação é necessário definir a intervenção e conjugar correctamente as interacções entre as pessoas, o processo aplicado, as características do produto e o projecto que orienta as actividades a desenvolver. É o que Roger Pressman apelida dos "quatro P's" associados ao desenvolvimento de software [Pressman00]. Só **pessoas** (informáticos, gestores e utilizadores) motivadas e comprometidas com o projecto garantem o respectivo sucesso; só um **processo** com técnicas e regras bem definidas permite atingir os objectivos propostos; só compreendendo as necessidades reais dos utilizadores se pode produzir um **produto** de qualidade; só com um **projecto** credível e controlado é possível cumprir prazos e custos propostos. Independentemente do método utilizado, estas deverão ser sempre preocupações comuns a todas as implementações de software.

Em 1996, Barry Boehm identificou sete questões que qualquer projecto de sistemas de informação deverá responder [Boehm96], e que são frequentemente agrupadas no que se designou pelo principio W5H2:

- Porque é que o sistema vai ser desenvolvido (W*hy*)?
- O que vai / deve ser feito (*What*)?
- Quando é que vai ser feito (*When*)?
- Quem é o responsável (*Who*)?
- Onde é que as responsabilidades estão localizadas (*Where*)?
- Como é que vai ser feito (*How*)?
- Quanto vai custar em termos de recursos (*How much*)?

Neste capítulo procuramos apresentar um conjunto de definições formais relacionadas com o desenvolvimento de software, cuja compreensão é fundamental, e são descritas em detalhe as actividades cuja realização, de um ponto de vista genérico, é expectável ao longo de todo o período que vai desde a identificação de uma necessidade de um utilizador até à produção do software que será a solução.

Gostaríamos desde já chamar a atenção do leitor para o facto da definição de vários dos conceitos apresentados não ser consensual. As definições apresentadas reflectem a nossa visão, que tem a preocupação de ser coerente e consistente, se basear em conhecimentos sólidos sustentados pela experiência prática, para além de procurar estar

alinhada com a opinião da maioria dos "pensadores" da Engenharia de Software.

2.2 Processos e Metodologias

Quando se fala do desenvolvimento de software, são frequentemente aplicadas expressões diferentes mas que muitas vezes representam a mesma ideia. É por isso fundamental a clarificação de alguns conceitos básicos adoptados neste livro, que se encontram, em geral, associados às áreas científica/tecnológica de “sistemas de informação” e de “engenharia de software”. Estes conceitos, de natureza técnica, são frequentemente mal entendidos e/ou mal aplicados. Estamos sobretudo a referir-nos aos conceitos de processo, metodologia e ciclo de vida.

O **processo de desenvolvimento de software** é um conceito de âmbito muito vasto, e pretende designar uma sequência de actividades, normalmente agrupadas em fases e tarefas, que são executadas de forma sistemática e uniformizada, que são realizadas por intervenientes com responsabilidades bem definidas, e que a partir de um conjunto de *inputs* produzem um conjunto de *outputs*. Um processo de desenvolvimento de software tem, segundo Booch, quatro objectivos fundamentais [Booch94]:

- Providenciar orientação sobre a sequência de realização das actividades envolvidas.
- Especificar os modelos descritivos do sistema que devem ser desenvolvidos.
- Dirigir as tarefas dos participantes e da equipa como um todo.
- Providenciar critérios para monitorização e avaliação dos modelos e actividades do projecto.

Nesta perspectiva, o conceito de **metodologia**, para além da sequência de etapas e procedimentos recomendados para serem aplicados durante o processo de desenvolvimento de sistemas de informação (ou seja, uma metodologia pressupõe a existência de um processo), acrescenta a esta definição a utilização de um conjunto de ferramentas, técnicas e notações [Booch94]. Para além disso, é normal que inclua ainda refe-

rências a diversos princípios e regras cujo objectivo é concretizar na prática as orientações mais teóricas que são expressas no conceito de processo, e nas quais podemos incluir:

- Regras de elaboração de estimativas (custos, prazos).
- Técnicas para efectuar medições.
- Procedimentos a seguir de forma a garantir a qualidade.
- Programas de formação.
- Modelos da documentação a produzir, vulgarmente designados por *templates*.
- Exemplos práticos detalhados.
- Técnicas para configuração da metodologia, que poderão ser aplicadas de modo a possibilitar a sua adaptação a realidades específicas.

De acordo com estas definições de processo e metodologia, o conceito de **ciclo de vida** pode ser encarado como um sinónimo de processo. A expressão ciclo de vida é mais antiga, aparecendo normalmente associada às abordagens tradicionais, enquanto o termo processo aparece sobretudo no contexto das abordagens mais recentes (ver discussão sobre o tema no Capítulo 3).

Gostaríamos aqui de realçar mais uma vez que estes três termos são muitas vezes utilizados indistintamente, com o significado que nós atribuímos ao conceito metodologia (o mais abrangente de todos). Sobretudo no caso das abordagens que se baseiam no paradigma da orientação por objectos (e em particular aquelas que utilizam UML para a modelação), o conceito mais utilizado é o de processo. Exemplos disso são o RUP (*Rational Unified Process*) e o ICONIX, ambos apresentados neste livro (Capítulos 10 e 11 respectivamente), e que se definem como processos, mas que segundo a definição por nós apresentada deverão ser considerados metodologias.

Aliás, é curioso verificar que segundo Philippe Kruchten, um processo de desenvolvimento de software é "um conjunto de passos parcialmente ordenados concebidos de forma a atingir um objectivo, que no caso da engenharia de software, é o de construir ou alterar um produto de software" [Krutchen00]. Como se vê não faz qualquer referência a técni-

cas de modelação ou ferramentas utilizadas, e que no entanto são descritas no âmbito do RUP, do qual é um dos principais mentores.

A preocupação com a utilização de abordagens sistemáticas foi talvez levada longe demais. Por exemplo, em meados da década 90 estimava-se que existiam mais de 1000 metodologias de desenvolvimento de sistemas de informação a serem utilizadas em todo o mundo [Jayaratna94]. Este número elevado não só dificulta a selecção e aplicação sistemática de uma metodologia, como é muitas vezes apontado como uma das razões pelo não atingimento do objectivo de uniformização (como há muitos standards, na prática não existe nenhum).

As metodologias actualmente existentes tiveram essencialmente duas origens distintas: (1) um primeiro grupo inclui as que evoluiram de experiências práticas, sobretudo das grandes consultoras de sistemas de informação, baseiam-se na aplicação de diversas técnicas, e normalmente deram origem a produtos comerciais; (2) um segundo grupo inclui aquelas que resultaram de esforços de investigação em universidades e que são frequentemente suportadas por conceitos teóricos muito próximos da matemática; por vezes, apenas se encontram descritas em livros e não são suportadas por ferramentas.

As metodologias são por natureza de âmbito geral, isto é, pretendem ser aplicadas em diferentes tipos de projectos, em diferentes indústrias e em várias organizações. Muitas das metodologias são relativamente rígidas, pois baseiam-se numa filosofia de desenvolvimento única com standards, técnicas e procedimentos uniformes, sem qualquer possibilidade de proceder a alterações ou configurações estruturais à própria metodologia. Para ultrapassar este problema houve organizações (sobretudo as mais ligadas à indústria de desenvolvimento de software) que criaram competências em diversas metodologias, de modo a poderem seleccionar a mais adequada para cada projecto concreto. Idealmente, as abordagens metodológicas deveriam ser flexíveis, de modo a permitirem a selecção de caminhos alternativos dentro de uma metodologia ou mesmo a configuração ou adaptação da própria metodologia.

2.3 Modelos e Modelação

Um **modelo** consiste na interpretação de um dado domínio do problema (fragmento do mundo real sobre o qual as tarefas de modelação e construção do sistema de informação incidem) segundo uma determinada estrutura de conceitos. Como exemplos de modelos podemos citar o modelo que resulta da análise de sistemas e o modelo de implementação.

Um **esquema** é a especificação de um modelo usando uma determinada linguagem, a qual pode ser formal ou informal (por exemplo, linguagem natural), textual ou gráfica. Quando a representação do esquema é gráfica, designa-se usualmente por **diagrama**. Como exemplos de esquemas e de diagramas temos o esquema relacional do modelo de dados de um sistema de crédito bancário ou o diagrama de classes de um sistema de facturação.

Um modelo é sempre uma interpretação simplificada da realidade. A ciência em geral, e a engenharia em particular, procurou desde sempre representar a "sua realidade" através de modelos mais ou menos correctos, mais ou menos abrangentes, mais ou menos detalhados. A ideia geral é que é preferível um mau modelo que nenhum modelo na descrição de qualquer sistema.

Outro aspecto relacionado é que um modelo apresenta apenas uma visão ou cenário de um fragmento do mundo real. É por conseguinte necessário a produção de vários modelos de forma a melhor representar e compreender o sistema correspondente Por exemplo, a construção de uma obra de engenharia civil apresenta inúmeros modelos, cada qual correspondendo a uma interpretação ou visão da mesma "realidade". As plantas, os alçados, as perspectivas ortogonais, as maquetes, o desenho da rede eléctrica, da rede de esgotos, da rede de água, da estrutura de betão, são diferentes representações ou modelos da mesma realidade.

A Figura 2.1 ilustra dois modelos complementares para uma mesma realidade: uma perspectiva (no lado esquerdo) e um alçado frontal (lado direito da figura) da casa.

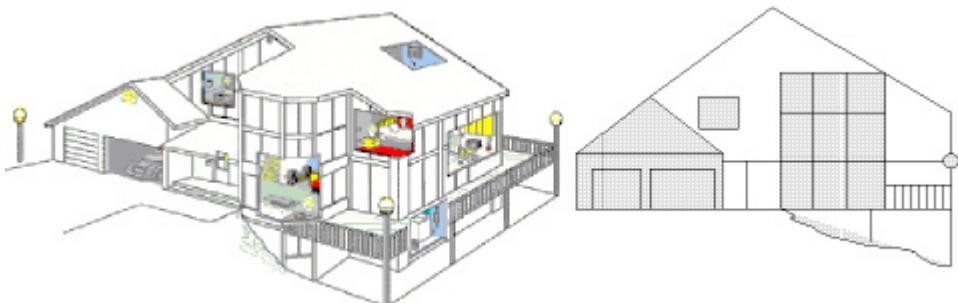



Figura 2.1: Diferentes modelos de uma mesma casa.

2.3.1 Importância da Modelação

A **modelação** (ou **modelização**) é a arte e ciência de criar modelos de uma determinada realidade. É uma técnica bem aceite e adoptada pela generalidade das disciplinas de engenharia conhecidas. Permite a partilha de conhecimento entre diferentes grupos de intervenientes (técnicos e não técnicos), facilita e promove a comunicação entre todos. Facilita a gestão mais eficaz e eficiente das equipas de projecto, ao dar uma visão mais adequada sobre os vários produtos a desenvolver, e permite ainda que as previsões de custos e prazos sejam efectuadas segundo critérios mais realistas o que também contribui para a minimização dos riscos associados.

A engenharia informática, embora sendo uma engenharia significativamente mais recente que a engenharia civil, necessita igualmente de adoptar notações e linguagens de representação dos seus modelos. Tal como nas restantes engenharias, também na engenharia informática se conseguem obter beneficíos através da modelação, que segundo [Booch99] incluem os seguintes:

- Os modelos ajudam a visualizar um sistema, quer seja a sua situação no passado, no presente ou no futuro.
- Os modelos permitem especificar a estrutura ou o comportamento de um sistema
- Os modelos permitem controlar e guiar o processo de construção do sistema.
- Os modelos documentam as decisões tomadas.

2.3.2 Princípios da Modelação

A experiência adquirida na actividade de modelação sugere que, independentemente do projecto, se verificam sempre um conjunto de quatro princípios básicos [Booch99].

P1: A escolha dos modelos a criar tem uma profunda influência no modo como o problema é encarado e consequentemente como a solução é obtida.

Isto significa que um modelo adequado (em termos de expressividade, abstracção ou abrangência) ilustrará de forma correcta determinados aspectos da construção de um sistema, oferecendo aos intervenientes no processo uma clarificação e simplificação dos mesmos. Por outro lado, um modelo inadequado terá como consequência uma inerente confusão e dificuldade de compreensão e comunicação.

P2: Cada modelo deve poder ser expresso em diferentes níveis de precisão/abstracção.

O grau de detalhe que um modelo deve apresentar está dependente do perfil do interveniente no projecto. Por exemplo, para o utilizador ou cliente de um sistema, apenas deverá ser relevante a perspectiva de utilização. Por outro lado, na perspectiva do analista e do programador do sistema, já será relevante a especificação de como é que o sistema deverá ser implementado. Por exemplo, se o sistema a considerar for um comando da televisão, o modelo para o utilizador final será uma descrição da sua utilização, enquanto que para o técnico, o modelo deverá especificar as mensagens trocadas entre diferentes aparelhos, tempos de atraso para garantir a boa recepção das mensagens, etc.

P3: Os melhores modelos reflectem a realidade.

Este princípio é um dos conhecidos “calcanhares de Aquiles” de algumas técnicas de modelação, em que existe uma manifesta separação entre os modelos utilizados para descrever "o que o sistema faz" e outros que detalham a forma "como o sistema faz", isto para já não falar na realidade que pretendem representar. É importante corrigir este problema, porque tal separação permite que a visão do sistema concebido e a visão do sistema implementado possam divergir significativamente.

P4: Nenhum modelo é suficiente por si só. Qualquer sistema não-trivial é representado de forma mais adequada através de pequeno número de modelos, razoavelmente independentes.

Tal como em engenharia civil, em que um prédio é representado por vários modelos complementares, também na engenharia de software ocorrem situações semelhantes. Na Parte 2 deste livro veremos que o UML se baseia significativamente neste princípio, ao propor várias notações para a produção de diferentes tipos de modelos. Note-se que o "razoavelmente independentes" significa que os modelos devem poder ser construídos de forma mais ou menos independente (até mesmo em paralelo), mas que deverão manter algum nível de integração, estilo, consistência e coesão entre si. É preciso não esquecer que o sistema objecto de modelação é comum a todos os modelos.

2.4 Boas Práticas no Desenvolvimento de Software

A complexidade é uma característica inerente ao software. Só um número muito reduzido de sistemas e tendo em conta o seu âmbito, número de pessoas afectadas ou criticidade da informação não apresenta esta característica (um bom exemplo disso é um sistema de gestão de contactos pessoais). Tal deve-se ao facto de, como Brooks sugeriu no seu famoso artigo *No Silver Bullet* [Brooks86], a complexidade do software é uma propriedade essencial, intrinseca à própria natureza do software, e não acidental, que ocorra esporadicamente. Segundo Booch [Booch94], esta complexidade tem origem em quatro factores:

- A complexidade do domínio do problema.
- A dificuldade de gerir o processo de desenvolvimento.
- A flexibilidade que é possível (ou não) implementar através de software.
- Os problemas de caracterizar o comportamento de sistemas discretos.

A incapacidade de controlar esta complexidade do software tem conduzido a projectos que ultrapassam os custos, os prazos e que não respeitam os requisitos definidos, tal como discutido no Capítulo 1. Courtois identificou vários atributos de um sistema complexo [Courtois85]:

- Um sistema complexo é composto por outros subsistemas relacionados, e assim sucessivamente, até se atingir um nível que é considerado elementar. Pode assim dizer-se que um sistema complexo é expresso através de uma hierarquia de elementos.
- A selecção dos componentes elementares de um sistema complexo é arbitrária e depende de quem a efectua, pois não existem critérios universais para o fazer.
- Num sistema complexo, com muitos elementos, as relações intra-componentes são mais fortes do que as inter-componentes.
- Cada subsistema é normalmente composto por poucos componentes diferentes.
- Um sistema complexo que funciona é invariavelmente uma evolução de um sistema simples que já funcionou; um sistema complexo concebido de raiz normalmente não funciona e dificilmente pode ser alterado de forma a que tal aconteça.

Conceito

Existe uma limitação natural da capacidade humana de lidar com a complexidade: não conseguimos estar em dois locais ao mesmo tempo, nem pensar em dois problemas simultaneamente. Por isso, Dijkstra sugeriu a aplicação do famoso princípio da **decomposição hierárquica**, também conhecido por *divide and conquer* ("dividir para conquistar"), através do qual um problema é dividido em sub-problemas mais elementares e assim sucessivamente, até que a sua resolução seja mais simples.

Conceito

Também a aplicação de um mecanismo de **abstracção** favorece a eliminação da complexidade: já que não é possível lidar com toda a realidade dos sistemas complexos, o ser humano opta por "esquecer" os detalhes menos importantes e focar a sua atenção nos mais relevantes, lidando com um modelo simplificado da realidade, mas considerado suficiente para entender e solucionar correctamente o problema em análise.

Para além destas duas ideias, da decomposição hierárquica e da abstracção, são frequentemente referidos na literatura outros princípios considerados fundamentais para a produção de software de qualidade, designadamente:

- O desenvolvimento deve ser efectuado de forma iterativa, repetindo as mesmas actividades em momentos temporais desfasados, mas detalhando o âmbito das funcionalidades do sistema (ver discussão na Secção 2.6.2).
- Efectuar uma gestão integrada dos requisitos, permitindo a verificação da rastreabilidade dos mesmos desde o momento da sua identificação até à respectiva implementação, facilitando todo o processo de gestão de alterações.
- Utilizar arquitecturas baseadas em componentes reutilizáveis, como forma de diminuir o esforço de desenvolvimento e posterior manutenção.
- Modelar o software através de diagramas gráficos, mais facilmente compreensíveis e menos sujeitos a interpretações subjectivas.
- Efectuar a verificação sistemática da qualidade, não apenas no final do desenvolvimento.
- Implementar um processo sistemático de controlo de alterações, de forma a gerir adequadamente um problema incontornável ("os requisitos de negócio mudam frequentemente") e a definir a forma e o momento em que essas alterações serão contempladas no sistema.

Cada uma destas melhores práticas tem impacto nas outras. Por exemplo, o desenvolvimento iterativo favorece a implementação de uma política de controle de alterações, uma vez que ao diminuir o tempo que vai desde a identificação da necessidade até à disponibilização de uma versão funcional (se bem que parcial) da aplicação, as alterações que entretanto ocorram podem ser incorporadas na nova iteração.

Existem outros princípios que deverão ser aplicados no sentido de garantir o sucesso no desenvolvimento de software:

- Conseguir e promover o envolvimento dos utilizadores.
- Utilizar uma abordagem orientada para a resolução de problemas.

- Definir e utilizar standards para o desenvolvimento e documentação.
- Justificar o desenvolvimento de software como uma actividade estratégica e como investimento financeiro.
- Conceber sistemas de modo a facilitar a sua expansão e alteração.

Vale a pena referir que existiram diversas iniciativas que ao longo do tempo foram desenvolvidas no sentido de melhorar o processo de desenvolvimento de software, nomeadamente:

- As iniciativas realizadas pelo *Software Engineering Institute*, que tem tido uma contribuição importante no estudo e definição de metodologias, estratégias e técnicas (para mais informações consultar [Paulk93] ou o endereço *www.sei.cmu.edu*).
- As iniciativas relacionadas com a implementação de políticas de qualidade, em particular da série 9000 da ISO [ISO91].
- A iniciativa SPICE (*Software Process Improvement Capability dEtermination*) [Dorling91].

A descrição destas iniciativas sai fora do âmbito deste livro, mas recomendamos a consulta das fontes acima apresentadas aos leitores mais interessados nestes temas.

2.5 Fases do Processo de Desenvolvimento de Software

Uma das ideias dominantes para a melhoria do desenvolvimento de software é a necessidade de aplicar um processo com fases bem definidas, que se subdividem em conceitos mais elementares (tarefas e actividades). Esta noção de processo começou a ser aplicada ao desenvolvimento de software a partir do momento em que se tornou óbvio que as iniciativas desenvolvidas ao nível das actividades de programação (como foram a programação estruturada) não eram suficientes para resolver os problemas que se levantavam no desenvolvimento de software, atendendo ao seu crescimento em termos de complexidade e dimensão.

Como vimos na Secção 2.2, a noção de processo consiste na definição de um conjunto de fases sequenciais, cada uma com tarefas bem definidas, nas quais participam pessoas com responsabilidades atribuí-

das e com diferentes competências, e que realizam diversas actividades; a natureza sequencial do processo implica que uma fase (bem como tarefa e actividade, consoante o nível de detalhe que estejamos a considerar) tenha que estar terminada antes de outra começar. Cada tarefa tem um conjunto de *outputs* bem definidos, que têm que ser produzidos antes que se possa considerar a tarefa como concluída.

Nesta perspectiva, uma fase é constituída por uma sequência de tarefas, e estas por sua vez são formadas por actividades. Os dois primeiros são conceitos abstractos, introduzidos de forma a agregar as actividades realizadas em função de critérios relativamente aos quais as actividades apresentam algumas semelhanças, como sejam objectivos a atingir, tipo de trabalho efectuado, relações e nível de inter-dependência. As actividades correspondem de facto a trabalho realizado, sendo pois a unidade mais elementar em que dividimos esta hierarquia de conceitos (se bem que alguns processos proponham ainda a divisão de uma actividade em passos mais elementares), e é aquela que pode ser medida e estimada em termos de planeamento. Em cada actividade são envolvidos diferentes membros de uma equipa, que desempenham distintos papéis, e são produzidos diferentes modelos com variados níveis de abstracção.

Na Figura 2.2 pretendemos mostrar a hierarquia de conceitos e a sua sequencialidade, de forma puramente abstracta e conceptual, e sem considerar nenhum processo concreto existente. No âmbito deste livro iremos considerar que um processo se pode dividir em três grandes fases ou momentos que traduzem a sua evolução temporal:

- **Concepção**, cujo objectivo é identificar "**o que é que o sistema deve fazer"**, nomeadamente a informação a processar, as funcionalidades a implementar, as restrições existentes, os critérios que determinam o sucesso e a aceitação; devem ainda ser consideradas e avaliadas diferentes alternativas e efectuada a respectiva selecção.
- **Implementação**, em que o objectivo é identificar "**o como fazer o sistema**", e construí-lo de facto; nomeadamente, serão definidas e construídas as estruturas de dados, os programas, os módulos, as interfaces (internas e externas), os testes a realizar; no final desta fase deverá ser disponibilizado o sistema de forma funcional.

- **Manutenção**, que inclui todas as alterações posteriores à aceitação do produto pelo cliente final: correcção de erros, introdução de melhorias e/ou de novas funcionalidades.

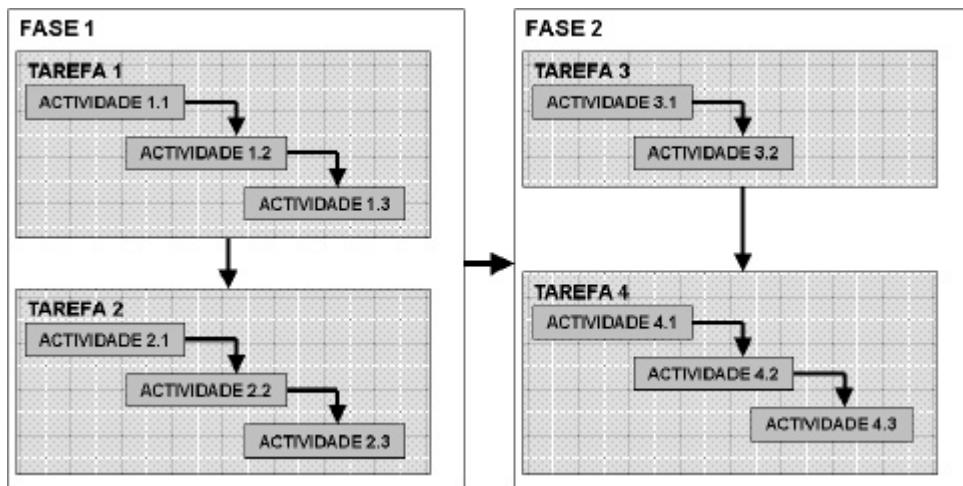
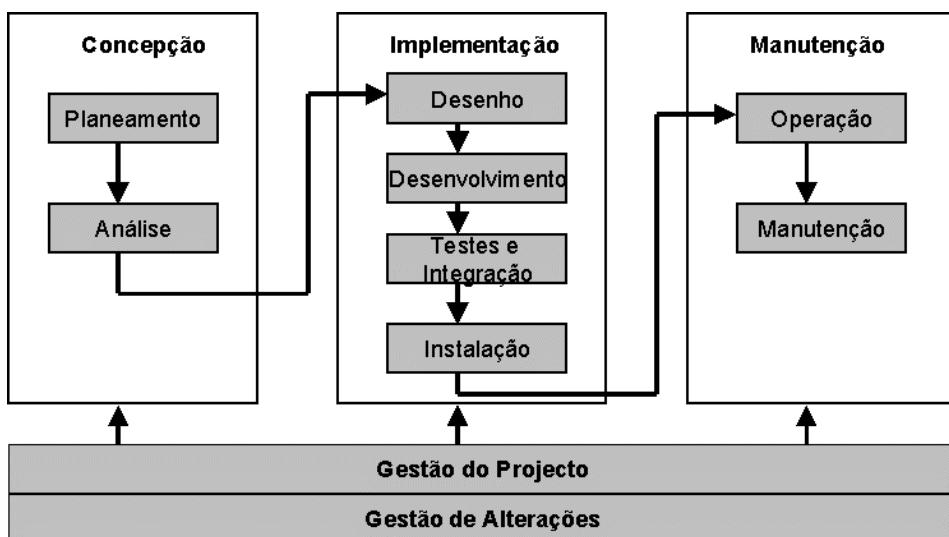



Figura 2.2: Representação genérica da hierarquia de conceitos fase, tarefa e actividade.

Se bem que a necessidade de aplicação de um processo estruturado ao desenvolvimento de software seja consensual, o mesmo não se pode dizer da sua definição, sobretudo na identificação das fases e tarefas que o compõem e respectiva sequência. Ao efectuar esta classificação temos como preocupação fundamental respeitar vários critérios lógicos (como sejam a semelhança entre os objectivos a atingir e o tipo de actividades realizadas), para além de procurarmos apresentar uma classificação sintonizada com a maioria dos autores da área da Engenharia de Software (veja-se por exemplo [Pressman00] ou [Schach99]). A sequência das fases considerada neste livro pode ser observada na Figura 2.3, onde representamos também o nível de detalhe das tarefas.

No entanto, não queremos perder a oportunidade de apresentar ao leitor algumas das diferenças mais relevantes que pode encontrar noutras classificações. Por exemplo, há quem considere que o levantamento dos requisitos do sistema e a respectiva especificação são tarefas distintas, enquanto outros autores juntam ambas na tarefa de análise e consideram-na como duas actividade da referida tarefa (esta foi a nossa opção, de acordo com os critérios acima apresentados).

Há também quem considere que deve existir uma tarefa de testes independente a seguir ao desenvolvimento do sistema, enquanto outros argumentam que o controle de qualidade e a realização de testes deve ser uma preocupação constante e realizada ao longo de todo o ciclo. Apesar de concordarmos com este princípio, optámos por considerar uma tarefa de testes independente, dado o volume e o esforço que os testes assumem no final da implementação, e também as especificidades destes testes relativamente ao controle de qualidade que é efectuado ao longo das restantes tarefas do ciclo.

Figura 2.3: Fases e tarefas do processo desenvolvimento de software.

Há ainda quem opte por considerar que o desenho é uma tarefa que deve ser colocada na fase de concepção, uma vez que as actividades realizadas são de definição e não propriamente de implementação. A nossa posição é que, apesar da validade desta observação, a tarefa de desenho tem como objectivo definir como se vai fazer, e nesse sentido deve ser colocada na implementação. Não é pelo facto da fase de implementação ser de natureza eminentemente prática e operacional que não podem existir tarefas e actividades de planeamento e de definição da mesma.

Tal como pode ser observado na Figura 2.3, as fases podem ser sub-divididas em tarefas mais específicas:

- **Planeamento,** correspondendo a uma identificação geral das necessidades, identificação e selecção de alternativas e definição de plano do trabalho.
- **Análise,** que inclui a identificação detalhada das funcionalidades do sistema (**Levantamento de Requisitos)** e a respectiva descrição (**Especificação do Sistema)** de modo a que os mesmos requisitos possam ser validados pelos utilizadores finais do sistema.
- **Desenho,** onde é realizada a definição detalhada da arquitectura global da solução (módulos, tabelas, interface, máquinas, etc.).
- **Desenvolvimento,** tarefa na qual é realizada a programação dos diversos componentes do sistema.
- **Testes (ou Integração),** em que o sistema no seu global é verificado com o objectivo de obter a aceitação do utilizador.
- **Instalação,** tarefa onde são executadas as actividades relacionadas com a disponibilização do sistema para os seus utilizadores finais, e que normalmente é designada por entrada do sistema em produção.
- Finalmente a **Manutenção,** o momento que corresponde ao tempo de vida útil do sistema e durante o qual serão efectuadas todas as alterações posteriores à entrada em funcionamento do produto.

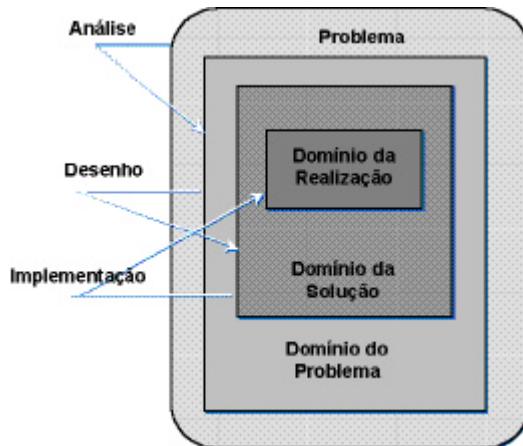



Figura 2.4: Do Problema para a Solução.

A Figura 2.4 apresenta uma visão diferente sobre o objectivo da aplicação de um processo de desenvolvimento de software: a partir do enunciado de um **problema** aplica-se um conjunto de actividades de **análise** para determinar o **domínio do problema**; a partir do **domínio do problema** aplica-se um conjunto de actividades de **desenho** para determinar o **domínio da solução**; a partir do **domínio da solução** aplica-se um conjunto de actividades de **implementação** para determinar o **domínio da realização**, que é o produto final de um projecto. Esta é uma visão muito formal, mas que traduz com rigor o que é de facto realizado.

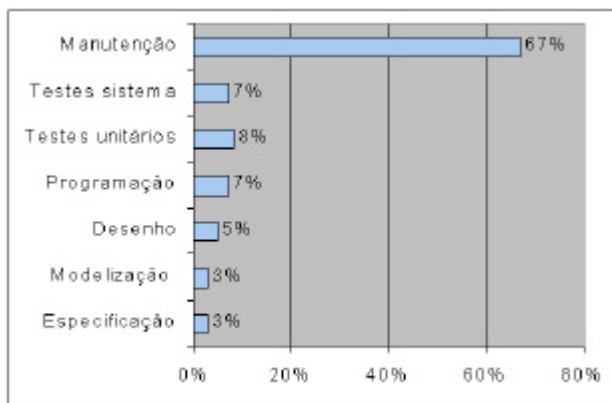



Figura 2.5: Custos relativos de diversas tarefas do processo de desenvolvimento de software.

Segundo diversos estudos realizados e condensados em [Schach99], os custos relativos de cada tarefa de um processo de desenvolvimento de software, tal como se verificavam em finais da década de 70, podem ser observados no gráfico da Figura 2.5.

Tal como foi referido no Capítulo 1, a manutenção do software sempre foi a tarefa do processo de desenvolvimento de software que maior esforço e custos relativos apresentava. Apesar das iniciativas de implementação de boas práticas no processo de desenvolvimento de software que se procuraram introduzir, esta proporção continua a ser idêntica [Yourdon96].

Passamos de seguida a analisar mais detalhadamente cada uma das tarefas do processo de desenvolvimento de software.

2.5.1 Tarefas Transversais

Existem algumas tarefas importantes, que agrupam actividades repetidas sistematicamente ao longo de todo o processo, ou parte deste, como é o caso da gestão do projecto ou a gestão de alterações.

Gestão do Projecto

A tarefa de gestão do projecto agrupa um conjunto de actividades relacionadas com a gestão de recursos humanos, de recursos financeiros e o controle dos prazos de execução das várias tarefas. É caracterizada essencialmente pelos seguintes grandes grupos de actividades: planeamento do projecto; controle e monitorização da sua execução; intervenção de modo a realizar medidas correctivas. É este nível que também funciona como a interface oficial para fora da equipa do projecto; o seu principal responsável designa-se por “gestor de projecto” e intervém directamente na elaboração dos diversos documentos de âmbito global e estratégico: descrição da visão global do negócio; glossário de termos do negócio; plano do projecto, avaliação do projecto; plano de gestão de alterações.

Gestão de Alterações

Em qualquer projecto de sistemas de informação, é fundamental que estejam previstos mecanismos de controle das alterações num processo em curso, já que tal como dizia o filósofo grego Heraclito, "a única certeza é a mudança permanente". A capacidade de monitorização das alterações e do seu impacto no sistema dá ao gestor do projecto o meio para avaliar e controlar a qualidade do mesmo. Áreas que apresentam alterações frequentes e imprevisíveis são normalmente consideradas áreas de risco e indiciam que a tarefa de análise foi realizada deficientemente.

2.5.2 Planeamento

Alguns autores não consideram o Planeamento como uma tarefa integrante do processo de desenvolvimento de software, incluindo as actividades por nós identificadas e englobadas nesta tarefa num âmbito de um planeamento global e estratégico dos sistemas informação da organização, e portanto não se enquadrando no âmbito da Engenharia de Software. Outros consideram-na como uma das tarefas horizontais e permanentes ao longo do processo, fazendo parte da tarefa de gestão de projectos. De facto, muitas das actividades e técnicas aplicáveis a esta tarefa são típicos de qualquer gestão de projectos.

Neste livro, nós consideramos uma tarefa de planeamento logo no início do processo, não só para apresentar a perspectiva mais abrangente, mas sobretudo porque é necessário ter uma visão global sobre o âmbito do trabalho a realizar de modo a elaborar um planeamento e a efectuar uma análise de viabilidade do projecto. Este tipo de abordagem pode significar que, a partir de um problema manifestado por um cliente, seja detectada a necessidade de desencadear vários projectos. A nossa perspectiva é que só temos um projecto depois do seu plano estar aprovado, e é esse o principal objectivo desta tarefa.

Existem ainda várias circunstâncias particulares em que esta tarefa deve ser realizada, por exemplo:

- Num projecto que envolve a selecção de entidades para posterior implementação do software, a elaboração de cadernos de encargos e a avaliação de propostas pode ser englobada nesta tarefa.
- Pode significar apenas uma investigação inicial, com recolha de informação suficiente sobre o problema ou oportunidade de modo a determinar se a situação actual justifica o desenvolvimento de uma solução suportada por um sistema de informação.

A grande preocupação desta fase é que a partir de um levantamento de alto nível das necessidades do negócio seja possível elaborar um plano do projecto a executar nas fases subsequentes, com identificação de actividades, recursos, prazos e custos. Para isso, existem um conjunto de actividades que deverão ser realizadas, designadamente:

- Compreender a missão e organização da empresa.

- Identificar e envolver todos interessados e afectados pela introdução do sistema.
- Obter uma visão de alto nível do funcionamento do sistema actual, caso ele exista.
- Definir o âmbito do sistema.
- Elaborar uma descrição de alto nível do problema.
- Identificar restrições, problemas e riscos do projecto.
- Identificar alternativas de implementação, proceder à sua avaliação e selecção.
- Apresentar resultados e recomendações, com justificação técnica, funcional e financeira.
- Elaborar e obter aprovação de um plano de projecto.
- Definir o processo de controlo do projecto.

Como se constata desta lista (não exaustiva) de actividades a realizar, muitas são actividades que qualquer gestor de projectos (independentemente da área do conhecimento humano em que trabalhe) deve saber executar. Por isso, também a maioria das técnicas a utilizar não são específicas dos sistemas de informação, mas comuns à área de gestão, tais como: análise financeira de custos e/ou benefícios; elaboração de estimativas; elaboração de planos de projectos (identificação de tarefas, elaboração de diagramas Pert e/ou Gantt); identificação de riscos e medidas de os controlar; elaboração de árvores ou tabelas de decisão; aplicação de técnicas de modelação de processos.

No final da tarefa, e como consta da lista de actividades, deverá ser obtida a aprovação para continuar com o projecto, sendo fundamental que esta concordância seja obtida de todos os interessados no projecto, em particular, do patrocinador do mesmo (o cliente/utilizador que tem mais interesse na resolução do problema e/ou que despoletou todo o processo), pela área de informática e muitas vezes (em função do volume de investimentos ou das políticas internas estabelecidas) por orgãos superiores da organização.

1 - **Introdução** (Âmbito e propósito do documento, Objectivos do projecto, Funções mais relevantes, Questões de performance, Restrições técnicas e de gestão)

2 - **Contexto do projecto** (Visão estratégica do negócio, Descrição de funções)

3 - **Especificações de alto nível dos requisitos**

4 - **Estimativas do projecto** (Dados históricos utilizados nas estimativas, Técnicas de estimação)

5 - **Riscos do projecto** (Análise de risco: Identificação, Estimação, Avaliação; Gestão do risco: Opções para evitar risco, Procedimentos de monitorização dos riscos)

6 - **Recursos do projecto** (Pessoas - Estrutura da equipa, Hardware e Software, Outros)

7 - **Calendarização** (*Work breakdown structure*, Diagramas de Pert e de Gantt)

8 - **Mecanismos de acompanhamento e controle**

9 - **Análise custos / beneficios**

10 - **Análise de alternativas**

O principal *output* desta tarefa será um plano de projecto (ver exemplo acima), que deverá reflectir sustentadamente a visão que se tem nesta fase do processo sobre as actividades a desenvolver futuramente, quer seja relativamente à sua inventariação, recursos, prazos e custos envolvidos, como também em relação aos benefícios potenciais que o sistema apresentará no futuro para toda a organização.

2.5.3 Análise

A tarefa de análise efectua o estudo detalhado do domínio do problema, e culmina na elaboração de um documento onde os requisitos funcionais da solução a implementar e outras questões relevantes (por exemplo, restrições, âmbito, fluxos de informação) são enumerados. Tem normalmente dois grandes momentos ou sub-tarefas (que tipicamente são realizados em simultâneo):

Levantamento de Requisitos

Um **requisito** é uma funcionalidade ou condição que o sistema deverá possuir. Para os identificar adequadamente, é aplicado um conjunto de técnicas de modo a obter a percepção detalhada daquilo que o sistema deverá efectuar. Estas poderão incluir a realização de reuniões com os interessados, a elaboração de questionários, a observação das actividades e do funcionamento do dia-a-dia, a recolha e análise de documentação diversa, a elaboração de pequenos protótipos do sistema que permitam validar mais facilmente a percepção obtida (seguindo o princípio que "uma imagem vale mais do que mil palavras"). Deve-se ter a preocupação de encontrar a melhor solução, pois às vezes aquilo que o utilizador pede não é sempre o que ele necessita (este facto está relacionado com o seu desconhecimento do que se pode obter de um sistema de informação). Outra questão a considerar tem a ver com a importância de identificar não apenas as funcionalidades actuais, mas sobretudo determinar a situação futura a atingir.

Especificação do Sistema

Contrariamente ao que se poderia julgar em primeira instância, definir e descrever os requisitos de um sistema não é tarefa fácil (como sabe qualquer indivíduo que desenvolveu um sistema de software): as interpretações e representações que cada indivíduo tem relativamente a um determinado sistema são quase sempre subjectivas (é a velha história do "copo meio vazio ou meio cheio"). O objectivo geral desta sub-tarefa é expressar sem ambiguidades o que o sistema deve fazer e não como fazer o sistema.

O conjunto de todos os requisitos e funcionalidades de um sistema é reunido num documento designado por "**Especificação de Requisitos**". Se bem a designação mais utilizada seja esta, tal não quer dizer que no relatório apenas estejam enumerados os requisitos. É normal incluir outro tipo de informação, nomeadamente os fluxos de informação que ocorrem no sistema, a identificação dos respectivos componentes e das suas relações, as restrições do sistema, as áreas internas e externas da organização afectadas.

Este documento deve ser visto como um contrato, daí a importância de ser rigoroso, objectivo, coerente e o mais completo possível. A especificação utiliza normalmente técnicas de modelação de processos, de definição de conceitos e de representação do comportamento do sistema, que serão apresentadas no próximo capítulo.

Tal como na tarefa anterior, no final desta tarefa de análise deve-se colocar a questão da continuidade do projecto, e obter das mesmas pessoas anteriormente referidas a aprovação da especificação de requisitos elaborada e do plano de projecto, agora revisto e detalhado.

2.5.4 Desenho

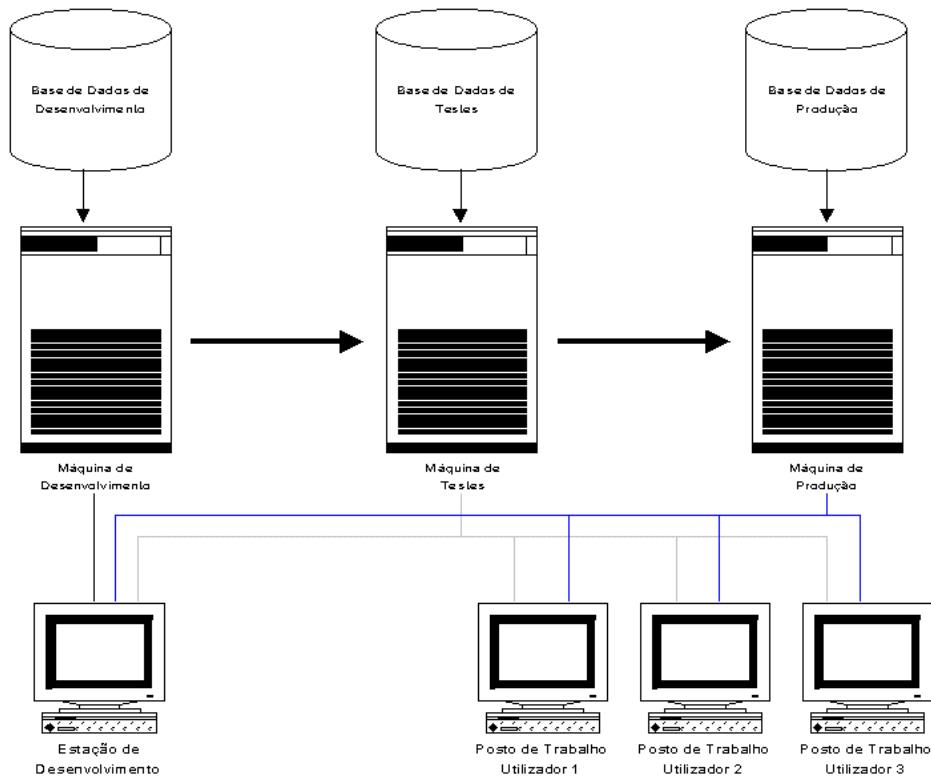
Como já vimos, o desenho pretende efectuar a definição do domínio da solução. Com base nos resultados produzidos pela tarefa de análise, procede-se à especificação, formal ou informal, das características que a implementação do sistema pretendido deverá apresentar, assim como a realização de determinadas optimizações consoante as características da infra-estrutura tecnológica de suporte. Esta inclui, por exemplo, arquitecturas de computadores, tecnologias de bases de dados, sistemas de execução de agentes, infraestruturas de objectos e/ou componentes, etc. É também nesta tarefa que deve ficar completamente definido o ambiente e as linguagens a utilizar no desenvolvimento.

Diz-se também com frequência que o desenho é a fase da especificação técnica, uma vez que são definidos os componentes aplicacionais (objectos, módulos, programas, servidores aplicacionais), tecnológicos (redes, máquinas, outros servidores) e os dados (estrutura de ficheiros e/ou de bases de dados, servidores a utilizar). Toda esta definição é realizada de forma integrada; é feita a descrição da lógica e dos algoritmos das aplicações, a interface e os outputs do sistema são também modelados.

Factores como o desempenho, a segurança e a operacionalidade do sistema deverão ser tidos em conta e utilizados (para além dos tradicionais critérios de custos e prazos) para seleccionar entre alternativas diversas. Devem também ser elaborados os planos de testes e de migração do sistema actual (ambos para executar numa fase posterior),

se bem que alguns autores protelem a elaboração destes planos para mais tarde, imediatamente antes da sua realização.

2.5.5 Implementação

A tarefa de implementação inclui todas as actividades de desenvolvimento do sistema propriamente dito, ou seja, que estão relacionadas com a concretização do modelo de desenho produzido na tarefa anterior. Os diversos componentes aplicacionais são codificados e testados de forma isolada, garantindo assim a respectiva correcção interna; este tipo de testes designa-se normalmente por testes unitários, uma vez que incidem sobre parcelas do sistema, e são realizados por cada programador de forma independente.

Preferencialmente, as actividades desta tarefa deveriam ser apoiadas por ferramentas, que a partir da especificação do modelo de desenho produzido seriam capazes de produzir automaticamente diversos componentes do sistema. Esta possibilidade pode ser concretizada por ferramentas e ambientes de desenvolvimento evoluídos e integrados, de forma que a tarefa seja executada com o menor esforço e no menor período de tempo possível. Actualmente assiste-se, no contexto dos sistemas de informação cliente/servidor, à crescente utilização de ambientes de desenvolvimento bastante produtivos (designados por ambientes RAD – *Rapid Application Development*), baseados em infra-estruturas de objectos/componentes evoluídos, complexos e providenciando paradigmas de prototipagem e desenvolvimento visual. No entanto, e apesar de todos os avanços verificados e de já existirem algumas ferramentas CASE com essas funcionalidades, a verdade é que a sua utilização não é em geral adoptada (ver Parte 4 deste livro).

A crescente aquisição pelas organizações de produtos previamente desenvolvidos com funcionalidades mais ou menos uniformizadas para a maioria das organizações (designados por pacotes), fez com que o tipo de actividades a executar ao longo desta tarefa tenha mudado de alguma forma: a programação pura cedeu uma parte da sua importância em favor de esforços de integração (entre os diversos componentes "pré-fabricados"), de parametrização (concretização de parâmetros genéricos previstos na aplicação para os valores utilizados pela organiza-

ção) e de customização adicional (adaptação de funcionalidades já existentes às especificidades da organização, podendo implicar desenvolvimento).

2.5.6 Testes

Para além dos já referidos testes unitários, executados durante a realização da tarefa anterior, esta tarefa destina-se à execução dos restantes tipos de testes. O objectivo é avaliar a adequada correcção e funcionamento de todos os componentes do sistema, principalmente os executáveis. A verificação consiste na confirmação que a codificação/ implementação do sistema está conforme (correcta) a especificação técnica produzida na fase de desenho, que por sua vez resulta dos requisitos especificados na análise. Por isso se diz que é importante efectuar a rastreabilidade dos requisitos.

Figura 2.6: Ambientes de desenvolvimento, testes e produção.

Estes testes deverão ser executados em condições idênticas aquelas que o sistema real irá possuir, mas fisicamente deverá ser suportado por outros componentes, de forma a que não existam interferências entre as actividades dos programadores e os testes dos utilizadores, e sobretudo com o trabalho normal do dia-a-dia destes últimos. Por isso, muitas organizações criaram uma separação mesmo ao nível físico entre os componentes de natureza aplicacional que estão a ser desenvolvidos, testados ou utilizados para suportar o negócio real da empresa, nomeadamente através do suporte em máquinas e ambientes distintos (ver Figura 2.6). O facto dos ambientes serem fisicamente separados não significa que não seja possível aceder a vários a partir do mêsmo posto de trabalho.

Os testes a realizar podem ser classificados em diferentes categorias, segundo as características do sistema que avaliam:

- Testes de carga: permitem analisar e avaliar o comportamento do sistema em situações de utilização intensiva e aferir as capacidades de escalabilidade.
- Testes de desempenho: permitem analisar o tempo de resposta do sistema e, dum forma geral, verificar o nível de utilização de recursos disponíveis.
- Testes de usabilidade: permitem analisar a adequabilidade do desenho das interfaces homem-máquina e constatar se o sistema é fácil de utilizar.
- Testes funcionais: permitem determinar a correcção da implementação de funcionalidades, conforme especificadas pelos correspondentes requisitos funcionais.

Os testes podem ainda classificar-se segundo o âmbito dos componentes do sistema que são alvo de verificação. Por exemplo:

- Testes unitários, já anteriormente referidos.
- Testes de integração: testes parcelares, que vários programadores realizam conjuntamente com vista a garantir que vários componentes interactuam entre si de forma adequada.
- Testes de sistema: testes globais em que todos os componentes do sistema são integrados; possibilitam a verificação da conformidade do sistema com todos os requisitos definidos.
- Testes de aceitação: testes formais que os utilizadores realizam sobre o sistema. Quando o sistema passa este (difícil!) teste, o cliente deverá aceitar o sistema como "pronto" e consequentemente este pode ser colocado em produção, ou operação, efectiva.

Nos casos em que o desenvolvimento pretende substituir um sistema existente, uma abordagem que muitas vezes é utilizada para a verificação do sistema consiste na repetição no novo sistema das actividades realizadas no sistema antigo, de forma a validar os resultados obtidos. Esta estratégia é designada por **paralelo de sistemas**.

A verificação pode ser apoiada a diferentes níveis por ferramentas de depuração (*debugging*) específicas e providenciadas conjuntamente e de forma integrada (ou não) com o resto do pacote do ambiente de desenvolvimento. Para além das ferramentas existem hoje em dia técnicas mais formais de realização de testes, cuja descrição vai muito

além do âmbito deste livro (para mais informação consultar [Pressman-00]).

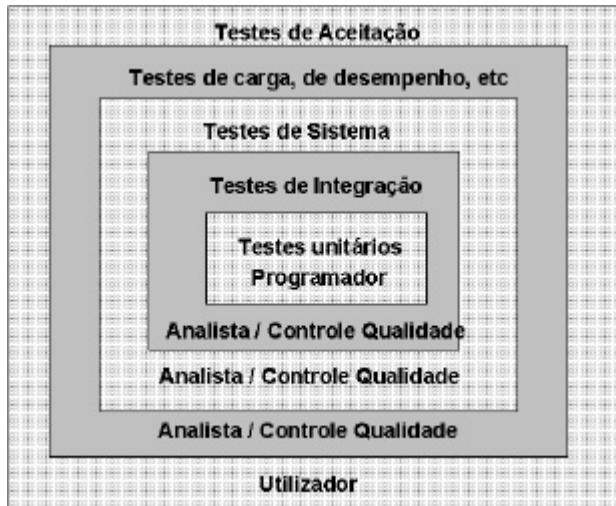



Figura 2.7: Tipos de testes e respectivo responsável.

O objectivo fundamental desta tarefa é conseguir a aceitação formal do cliente relativamente à adequação do sistema às suas necessidades, e a aprovação pelo mesmo da decisão de colocar o sistema em funcionamento.

2.5.7 Instalação

A tarefa final da fase de implementação do sistema é a respectiva instalação em ambiente de produção, de forma a disponibilizar as funcionalidades concebidas para os seus verdadeiros utilizadores. Esta tarefa consiste na preparação e instalação do sistema na infra-estrutura computacional destino e na sua subsequente operação regular. Envolve um conjunto nem sempre entendido e quantificável de tarefas, que muitas vezes são esquecidas na altura da preparação do plano do projecto, tais como:

- Configuração e parametrização do sistema implementado bem como dos sistemas de suporte requeridos (por exemplo, sistemas de gestão de bases de dados, servidores aplicacionais, etc.).
- Instalação dos componentes aplicacionais necessários.

- Definição de perfis, de utilizadores e de níveis de segurança.
- Formação dos utilizadores no sistema.
- Elaboração de documentação de operação, administração e utilização do sistema.
- Definição e implementação de políticas de segurança (por exemplo, criação de cópias).

Caso exista já um (ou mais) sistemas em funcionamento é necessário efectuar a migração para o novo sistema, de acordo com o que foi definido no plano anteriormente elaborado. Num momento previamente acordado é efectuada a migração para o novo sistema, passando os utilizadores a partir deste momento a trabalhar no novo ambiente, designado de produção.

2.5.8 Manutenção

Durante a vida útil de qualquer sistema de software são detectados problemas que não são devidamente verificados durante a fase de implementação (designados por *bugs*). Surgem também inúmeras solicitações internas e externas relativamente a pedidos de alteração de requisitos que não foram contemplados originalmente na fase de concepção, e que exigem a elaboração de novas versões/actualizações do software. Durante o tempo de vida útil do software são ainda detectados problemas que apenas podem ser identificados nesta fase; estão normalmente relacionados com questões de desempenho do sistema e apenas se tornam perceptíveis com a sua crescente utilização. A Figura 2.8 mostra a proporção em que estas diversas situações ocorrem.

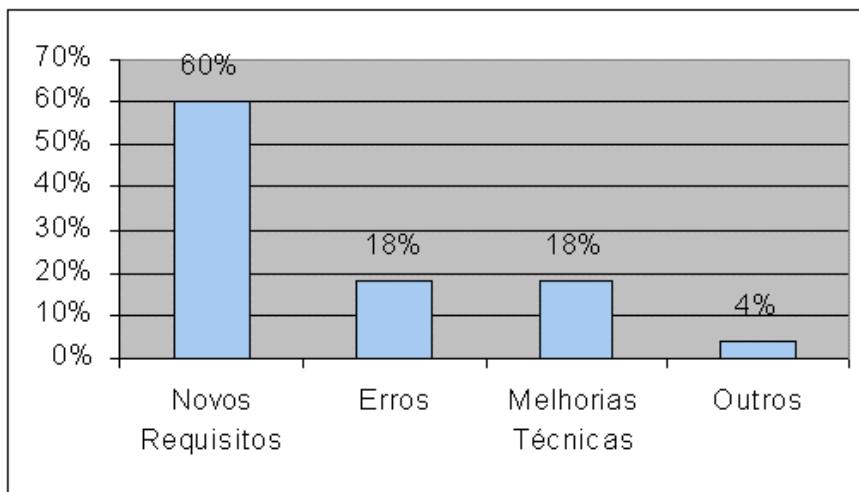



Figura 2.8: Percentagem relativa das intervenções que ocorrem durante a manutenção do software.

O objectivo da tarefa de manutenção de software é garantir que a ocorrência de alguma destas situações é convenientemente tratada. Actualmente, muitos autores não veem a manutenção como uma tarefa com actividades próprias, mas sim como o período que se inicia imediatamente após a entrada em produção do sistema, e que dura enquanto o software se mantiver a funcionar. Isto porque as actividades a executar sobre o sistema fazem de facto parte de outras tarefas já descritas (análise, desenvolvimento, testes, etc). Por isso, é mais correcto considerar que a manutenção desencadeia uma nova iteração de todo o processo de desenvolvimento de software, perspectiva com a qual nós concordamos.

Complementarmente à manutenção são realizadas outras actividades que garantem o bom funcionamento do sistema segundo diversos critérios, e que têm uma intervenção sobretudo a nível tecnológico. Estamos a referir-nos ao conjunto de actividades que pode ser genericamente designado por operação do sistema, e que inclui, entre outras, a realização de cópias de segurança do sistema, a verificação dos parâmetros de desempenho, a definição de novos utilizadores, etc.

2.6 Processos de Desenvolvimento de Software

A descrição genérica das fases e tarefas dos processos efectuada na secção anterior é concretizada em modelos em que a sequência e os nomes das fases e tarefas assumem diversas especificidades. Independentemente das particularidades de cada processo, podemos distingui-los genericamente segundo duas grandes aproximações: aqueles que seguem uma aproximação segundo um modelo em cascata e os que têm uma aproximação iterativa. Note-se que esta distinção é ortogonal ao facto do processo usar uma abordagem estruturada ou orientada por objectos, conforme se poderá verificar pela apresentação a efectuar no Capítulo 3. Pode-se ter, por exemplo, um processo iterativo adoptando uma abordagem estruturada, ou um processo em cascata adoptando uma abordagem orientada por objectos, ou qualquer outra combinação. Enquanto a iteratividade tem a ver com a sequência das actividades, a abordagem tem a ver com o paradigma, os conceitos base e a forma de modelar e solucionar o problema.

2.6.1 Processos em Cascata

Os processos tradicionais de desenvolvimento de software são caracterizados por adoptarem um **modelo em cascata** (ver Figura 2.9), em que as actividades a executar são agrupadas em tarefas, executadas sequencialmente, de forma que uma tarefa só tem início quando a tarefa anterior tiver terminado.

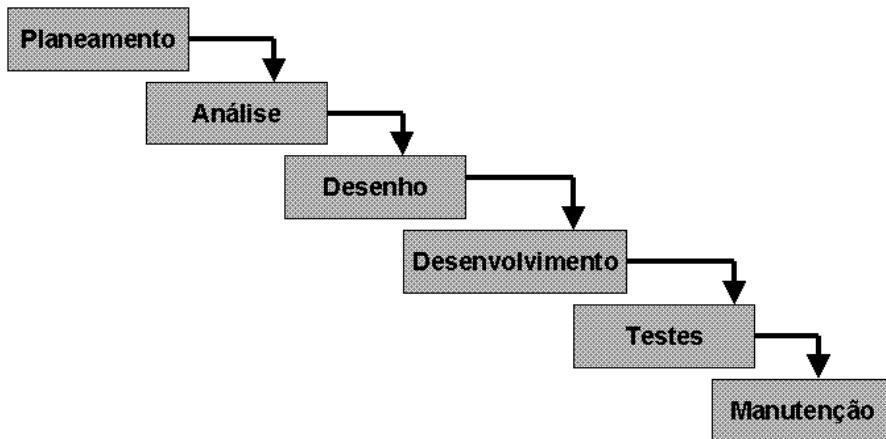



Figura 2.9: Processo em cascata.

O modelo em cascata tem a vantagem que só se avança para a tarefa seguinte quando o cliente valida e aceita os produtos finais (documentos, modelos, programas) da tarefa actual. Este modelo baseia-se no pressuposto que o cliente participa activamente no projecto e que sabe muito bem o que quer. Em situações de desenvolvimento por uma empresa de software responsável pelo projecto existe a garantia, ou defesa formal, que o cliente aceitou os referidos produtos. De facto, esta salvaguarda formal dá uma significativa protecção à empresa de software quando, como é corrente, o cliente vem dizer que afinal não é "aquilo" que necessita. Mas, infelizmente, esta salvaguarda não evita que o projecto não obtenha os resultados originalmente esperados, que o cliente fique desapontado, e que se tenha desperdiçado tempo e recursos indevidamente.

O modelo em cascata apresenta ainda outras limitações. Por exemplo, promove a compartimentação dos esforços ao longo das diferentes tarefas e consequentemente desencoraja a comunicação e partilha de visões entre todos os intervenientes do projecto, por exemplo, entre os analistas, os responsáveis pelo desenho, os programadores, e os utilizadores. Minimiza o impacto da compreensão adquirida no decurso de um projecto, uma vez que se um processo não pode voltar atrás de modo a alterar os modelos e as conclusões das tarefas anteriores, é normal que as novas ideias sobre o sistema não sejam aproveitadas.

De forma a eliminar este problema foi definido um novo processo, baseado no modelo clássico em cascata, e designado por **modelo em cascata revisto** (ver Figura 2.10), que prevê a possibilidade de a partir de qualquer tarefa do ciclo se poder regressar a uma tarefa anterior de forma a contemplar alterações funcionais e/ou técnicas que entretanto tenham surgido, em virtude de um maior conhecimento que se tenha obtido. O risco desta abordagem é que, na ausência de um processo de gestão do projecto e de controle das alterações bem definido, podemos passar o tempo num ciclo sem fim, sem nunca se atingir o objectivo final que é disponibilizar um sistema a funcionar.

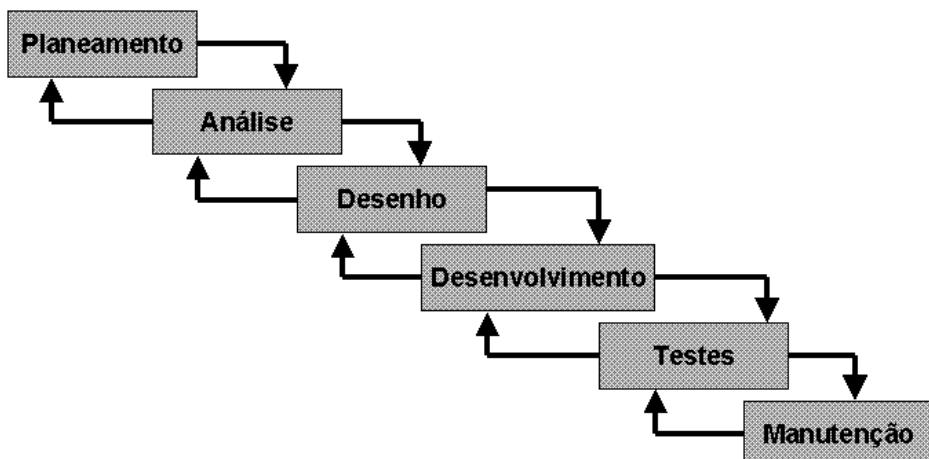



Figura 2.10: Processo em cascata revisto.

Ambos os modelos anteriores apresentam o problema da extensão do período de tempo que decorre entre o início do projecto e a entrega do produto final, que pela sua duração não corresponde minimamente às expectativas do cliente. Uma solução encontrada para resolver parcialmente este problema consistiu na aplicação de técnicas de prototipagem logo no início do processo, sendo este modelo designado por **prototipagem rápida**. Não elimina a necessidade de todas as restantes tarefas sequenciais, mas permite que o cliente veja e valide um modelo da interface (e das funcionalidades) que irá ter disponível, reduzindo também os problemas da comunicação.

Como não há "rosas sem espinhos", é preciso neste caso ter um cuidado acrescido de forma a não ceder às naturais pressões do cliente, no sentido deste passar a utilizar de imediato o protótipo. É que o protótipo é normalmente apenas constituído por uma sequência de ecrãs sem quaisquer validações ou acesso a bases de dados, e a disponibilização destas funcionalidades requer tempo e um esforço significativo. A satisfação da pretensão do cliente significaria comprometer a qualidade do produto em detrimento de "ter qualquer coisa a funcionar". Existe ainda o perigo de se perder a visão mais global e abrangente do sistema e respectivas funcionalidades em detrimento de uma visão mais restrita e imediatista baseada em sequências de ecrãs.

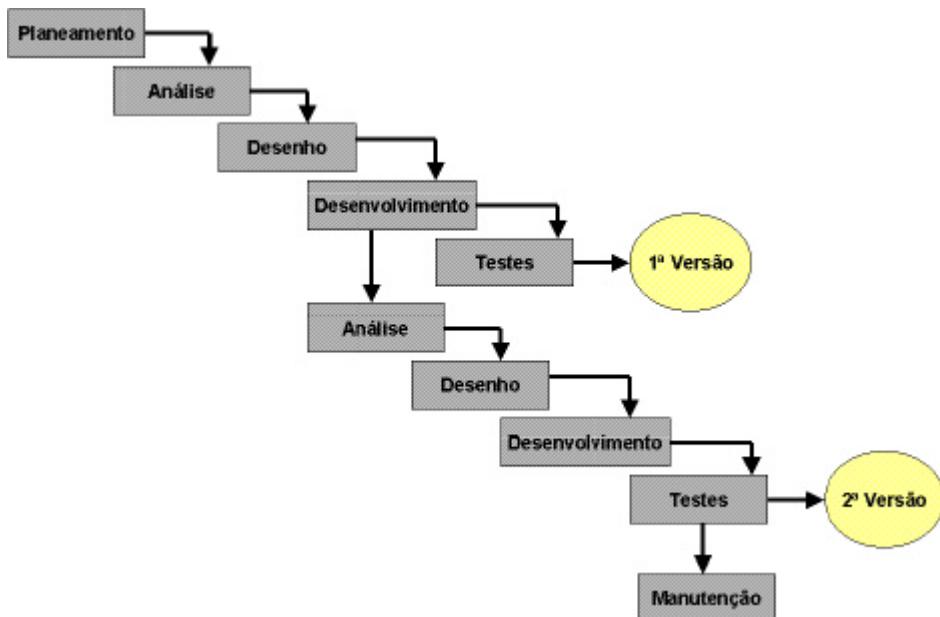
2.6.2 Processos Iterativos e Incrementais

Em contraste com os processos baseados no modelo em cascata, os processos mais recentes de desenvolvimento de software promovem na sua generalidade o modelo iterativo e incremental. Com estas ideias encoraja-se a comunicação entre todos os intervenientes de modo a produzir sistemas finais mais robustos e com qualidade superior. De certa maneira, o processo em cascata revisto é um precursor destes processos iterativos.

Iterativo

A noção de **processo iterativo** corresponde à ideia de "melhorar (ou refinar) pouco-a-pouco" o sistema. Um excelente exemplo de aplicação do processo iterativo encontra-se no trabalho artístico, em que o resultado final de uma obra de arte sofre inúmeras iterações (algumas das quais, por vezes, destrutivas para o seu resultado final). O âmbito do sistema não é alterado, mas o seu detalhe vai aumentando em iterações sucessivas.

Para além desta característica, o desenvolvimento iterativo apresenta ainda outras vantagens significativas para o desenvolvimento de software [Kruchten00]:

- Os riscos e dúvidas com maior importância são identificados no início do processo, nas primeiras iterações, quando é possível tomar medidas para os corrigir.

- Esta abordagem encoraja a participação activa dos utilizadores de modo a identificar os verdadeiros requisitos do sistema.
- A execução de testes contínuos e iterativos permite uma avaliação objectiva do estado do projecto.
- As inconsistências entre a análise, o desenho e a implementação são identificadas atempadamente.
- O esforço dos diversos elementos da equipa é distribuído ao longo do tempo.
- A equipa pode aprender com experiências anteriores e melhorar continuamente o processo.
- Pode ser demonstrado inequivocamente a todos os envolvidos e interessados no projecto o respectivo avanço.

Incremental

A noção de **processo incremental** corresponde à ideia de "aumentar (ou alargar) pouco-a-pouco" o âmbito do sistema. Uma boa imagem para este atributo é a de uma mansão que foi construída por sucessivos incrementos a partir de uma primeira casa com apenas duas divisões.

Iterativo e Incremental

O princípio subjacente ao **processo iterativo e incremental** (ver Figura 2.11) é que a equipa envolvida possa refinar e alargar pouco-a-pouco a qualidade, detalhe e âmbito do sistema envolvido. Por exemplo, numa primeira iteração deve-se identificar a visão global e determinar a viabilidade económica do sistema; efectuar a maior parte da análise e um pouco de desenho e implementação. Numa segunda iteração, deve--se concluir a análise, fazer uma parte significativa do desenho, e um pouco mais de implementação. Numa terceira iteração, deve-se concluir o desenho, fazer-se parte substancial da implementação, testar e integrar um pouco; etc. Ou seja, a principal consequência da aproximação iterativa é que os produtos finais de todo o processo vão sendo amadurecidos e completados ao longo do tempo, mas cada iteração produz sempre um conjunto de produtos finais.

Figura 2.11: Processo iterativo e incremental.

Existem ainda outras vantagens adicionais dos modelos iterativos e incrementais, por oposição ao modelo em cascata, e entre estas podem-se salientar as seguintes: possibilidade de avaliar mais cedo os riscos e pontos críticos ou mais significativos do projecto, e identificar medidas para os eliminar ou pelo menos controlar; definição de uma arquitectura que melhor possa orientar todo o desenvolvimento; disponibilização natural de um conjunto de regras para melhor controlar os inevitáveis pedidos de alterações futuras; e permite que os vários intervenientes possam trabalhar mais efectivamente pela interacção e partilha de comunicação daí resultante.

Finalmente uma última nota. É normal encontrarmos na literatura apenas referências ao termo iterativo, mas em muitas situações a definição que lhe está subjacente implica também a noção de processo incremental, no sentido que é apresentada neste livro.

Espiral

O **modelo em espiral** é uma pequena variante do modelo iterativo e incremental. Foi proposto por Barry Boehm [Boehm88] como resposta

às críticas que os processos existentes na altura não favoreciam a utilização de prototipagem e reutilização de software. Por isso, e para além das tarefas e actividades previstas pelos outros processos, propõe logo de seguida à tarefa de planeamento a realização de uma tarefa de prototipagem e de análise do risco, como forma de eliminar os principais problemas e identificar os requisitos do sistema.

2.7 Conclusão

O objectivo deste capítulo foi o de realçar a importância da aplicação de métodos e técnicas que contribuam para a melhoria do processo de desenvolvimento de software, nomeadamente a aplicação dos princípios da abstracção e da decomposição hierárquica face à complexidade e dimensão crescentes do software, e a utilização de um processo de desenvolvimento uniformizado.

Foram apresentados diversos conceitos relacionados com o processo de desenvolvimento de software sempre numa perspectiva genérica e abstracta. Foi o que aconteceu com as definições dos conceitos de metodologia e processo, e assim como a discussão relativa às noções de fases, tarefas e actividades. Houve a preocupação de efectuar esta apresentação sem estar condicionado por nenhuma abordagem metodológica concreta.

No próximo capítulo vamos analisar como é que ao longo do tempo estes conceitos têm evoluído, e perceber o contexto em que surgem as abordagens orientadas por objectos, e notações e linguagens de modelação como é o caso do UML.

2.8 Exercícios

Ex.8. "O conceito de metodologia concretiza, põe em prática, o conceito teórico de processo". Comente esta afirmação.

Ex.9. Indique algumas razões pelas quais a estrutura de um processo de desenvolvimento tradicional é criticada hoje em dia.

Ex.10. Muitas metodologias defendem actualmente a importância de se aplicarem sistematicamente técnicas de controlo de qualidade, e de se realizarem testes ao longo de todas as fases do ciclo de vida, e não apenas posteriormente à programação. Indique de que modo é que estas actividades poderão ser concretizadas nesses outros momentos.

Ex.11. Indique de que maneira é que o desenvolvimento iterativo pode contribuir para favorecer a aplicação sistemática de técnicas de controlo de alterações.

Ex.12. Na sequência das críticas apontadas ao ciclo de vida em cascata, foi sugerida a aplicação de técnicas de prototipagem como forma de ultrapassar esses problemas, o que resultou num novo ciclo de desenvolvimento. Concorda que, apenas por este facto, se possa considerar um novo ciclo? Justifique.

Ex.13. Quais as diferenças que existem entre as tarefas de análise e de especificação de requisitos? Estas duas tarefas são sempre independentes uma da outra, ou existem abordagens em que estão incluídas na mesma tarefa?

Ex.14. "O modelo em espiral não traz nenhum valor acrescentado aos processos iterativos e incrementais". Comente esta afirmação.

Ex.15. Relativamente às tarefas de um processo de desenvolvimento de software apresentadas na Figura 2.3, existirão situações em que algumas possam ser eliminadas? Justifique a sua afirmação.

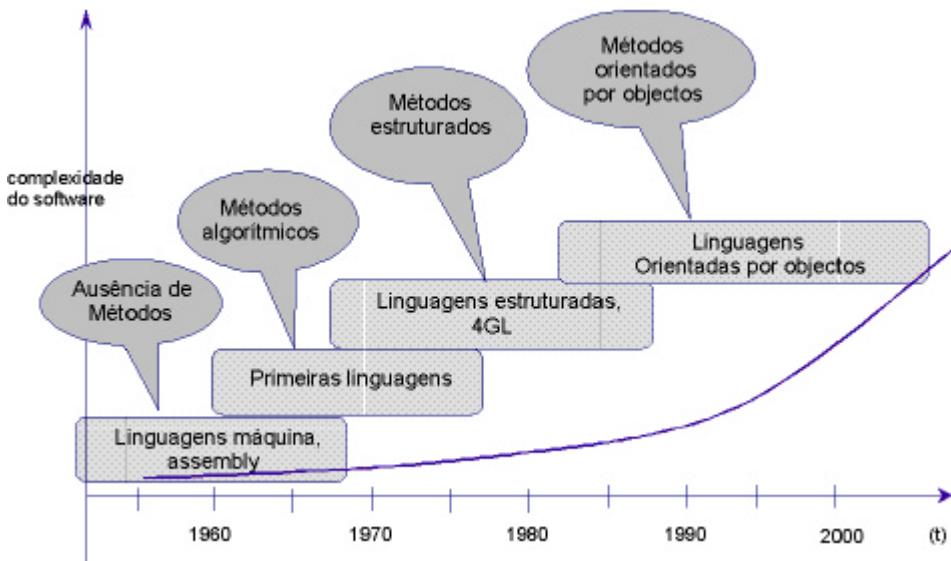
Capítulo 3 - Evolução das Metodologias de Desenvolvimento de Software

Tópicos

- Introdução
- A Programação como Fonte de Inovação
- Desenvolvimento Ad-Hoc
- As Metodologias Estruturadas
- As Metodologias Orientadas por Objectos
- Outras Metodologias
- Comparação de Metodologias
- Conclusão
- Exercícios

3.1 Introdução

No início dos anos 60, Edsger Dijkstra afirmou que qualquer pedaço de código é essencialmente uma série de instruções de natureza matemática, e como tal é possível provar a sua correcção [Dijkstra65]. Posteriormente, em 1969 no artigo intitulado *Structured Programmming* [Dijkstra-69], ele realçou a importância da preocupação com a prevenção de erros, para além de utilizar pela primeira vez a expressão "programação estruturada".

Como já referimos no Capítulo 1, a expressão engenharia de software foi pela primeira vez aplicada em 1967 numa conferência da NATO, e em 1971, Niklaus Wirth publicou o artigo "*Program Development by Stepwise Refinement*" [Wirth71], com base no trabalho de Edsger Dijkstra e Barry Bohm [Bohm66], defendendo a ideia da construção de programas de forma incremental, à custa de unidades mais elementares. Em 1972, David Parnas publicou o seu célebre artigo sobre encapsulamento de informação [Parnas72], no qual reforçava a ideia de Wirth relativamente à importância de dividir um programa em unidades mais elementares, cada uma delas apenas disponibilizando um conjunto de funcionalidades, mas "escondendo" a respectiva implementação.

Todos estes exemplos reforçam a importância de conceitos propostos desde finais da década de 60 e princípios da década de 70 e que tiveram impacto na actividade e técnicas de programação utilizadas. No entanto, já em finais dos anos 60 se reconhecia que este esforço não seria suficiente, e que era preciso rentabilizar o trabalho do programador através de medidas com impacto em áreas que não apenas a programação.

Larry Constantine tentou identificar mecanismos que possibilitassem que a concepção inadequada de software fosse evitada desde o início, aquando da identificação de requisitos. Em conjunto com Wayne Stevens e Glen Myers propôs o conceito de *Composite Design*, renomeado para "desenho estruturado" [Stevens74], que procurava representar sob a forma de diagramas as acções de um programa que posteriormente seriam codificadas. Esta foi uma das primeiras iniciativas no sentido de definir um processo mais consistente e abrangente, aplicado a todo o ciclo de desenvolvimento de software e cujos conceitos foram introduzidos no Capítulo 2.

O objectivo deste capítulo é apresentar, de forma resumida, a evolução histórica dos processos de desenvolvimento de software, de forma a enquadrar e compreender a importância dos esforços recentes em torno do paradigma da orientação por objectos e em particular da linguagem UML. Apresentamos e discutimos com particular relevância os processos que adoptam uma abordagem estruturada (Secção 3.4) e orientada por objectos (Secção 3.5).

3.2 A Programação como Fonte de Inovação

Até muito recentemente os principais saltos qualitativos em termos de conceitos com impacto significativo nas áreas abrangidas pela engenharia de software iniciaram-se sempre pela área das linguagens de programação.

Figura 3.1: Visão histórica dos métodos de desenvolvimento de software.

A Figura 3.1 ilustra a evolução das principais técnicas e abordagens que ao longo do tempo têm ocupado lugar de destaque ao nível da programação, consoante o grau de complexidade e da dimensão do software. De acordo com estes parâmetros, podemos identificar quatro fases principais.

Num primeiro momento surgiram as linguagens de baixo nível, muito próximas da tecnologia e fortemente condicionadas por ela, na qual se incluem as linguagens máquina, em que as instruções eram meras sequências de bits. A primeira iniciativa no sentido de elevar a actividade de programação surgiu com o aparecimento das linguagens *assembly*, que utilizavam mnemónicas como forma de substituir o código máquina.

A primeira alteração com impacto deu-se com o aparecimento das primeiras linguagens simbólicas (Fortran, Algol, Cobol), numa tentativa de aproximar mais as técnicas de programar do processo de raciocínio humano. As primeiras linguagens eram utilizadas sobretudo para cálculos científicos e de engenharia (Fortran), mas gradualmente foram ganhando peso as linguagens para aplicações comerciais (Cobol). Eram linguagens eminentemente imperativas, em que as instruções diziam o que se fazia. No entanto, a grande ausência de regras sintácticas e semânticas possibilitava a construção de programas difíceis de compreender. Data desta altura a expressão "código esparguete", que resultou da utilização abusiva da instrução GOTO, e que resulta dos inúmeros saltos que um programa poderia apresentar!). Esta segunda fase decorreu principalmente durante os anos 1960/70, e caracterizou-se pelo desenvolvimento de aplicações razoavelmente simples e de pequena dimensão, mas em que as ferramentas de apoio eram ainda relativamente incipientes, e os métodos de desenvolvimento eram essencialmente algorítmicos.

A terceira fase, que se iniciou durante os anos 70, foi caracterizada pelo desenvolvimento de sistemas de maior dimensão com base em linguagens estruturadas de mais alto nível (por exemplo, Pascal, C, 4GL), que eram suportadas por métodos estruturados.

Os programas construídos com recurso às linguagens estruturadas apresentavam uma organização em blocos de código, constituídos à custa de construções simples (isto é, sequências, selecções e repetições). O aumento da complexidade e dimensão dos programas causou graves problemas nas técnicas de programação, pois os programas eram constituídos por uma sequência de blocos de instruções, todos localizados fisicamente no mesmo ficheiro, o que não facilitava a decomposição do problema noutros mais elementares nem suportava adequadamente o trabalho em equipa. O outro grande problema residia na utilização abusiva dos dados globais a um programa, que possibilitavam que todas as instruções de um programa acedessem a esses dados, tornando difícil perceber quem manipulava que informação, e dificultando a verificação da consistência do sistema.

Foi nesse sentido que a linguagem Modula-2 (definida pelo mesmo autor da linguagem Pascal, Niklaus Wirth, e descrita de forma acessível

em [Walmsley90]) introduziu o conceito de módulo, enquanto bloco de código no qual um programa se poderia decompor. A ideia era agrupar num módulo dados e código, que estivessem relacionados de forma a limitar as interacções entre módulos à invocação de funções. No entanto era ainda possível aceder do exterior aos dados declarados dentro de um módulo.

De forma a impossibilitar estes acessos, os teóricos propuseram o conceito de tipo de dados abstracto, cuja principal concretização prática ocorreu na linguagem ADA [Booch93]. Estes novos tipos de dados permitem "esconder" totalmente partes da sua implementação (quer ao nível dos dados quer do código), e disponibilizam para o exterior o que se designa por interface. Pela primeira vez, foi possível definir uma associação tão forte entre dados e código.

A evolução dos conceitos referidos nos dois parágrafos anteriores em termos de abstracção e de encapsulamento da informação conduziu naturalmente ao aparecimento das linguagens orientadas por objectos (por exemplo, Smalltalk, C++, Java, Object Pascal). A principal distinção entre os objectos e os tipos de dados abstractos é o suporte à noção de herança pelos primeiros, o que não se verifica nos segundos (ver Secção 3.5). Nesta fase, o desenvolvimento de software é ainda mais complexo e de maior dimensão, o que exige o suporte por parte de tecnologias avançadas como sejam *frameworks* aplicacionais, *frame-works* de *middleware*, componentes de software e utilização de ambien-tes de desenvolvimento.

Na Figura 3.2 é apresentado um modelo simplificado que demonstra a visibilidade dos dados e como estes se encontram relacionados com o código, relativamente a cada um dos conceitos acima referidos: (1) programa, (2) módulo, (3) tipos de dados abstractos e (4) objectos.

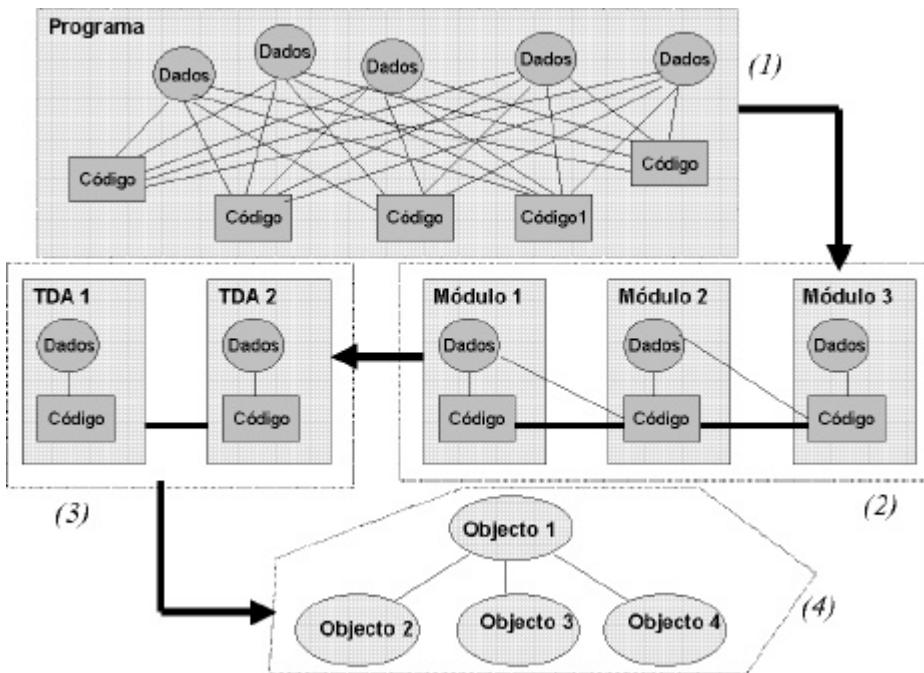



Figura 3.2: A evolução dos conceitos de abstracção nas linguagens de programação.

Na Figura 3.3 pode ser observado um diagrama que resume a evolução das linguagens de programação nos últimos 40 anos, e onde apenas representamos as linguagens mais significativas para as abordagens de desenvolvimento discutidas neste livro. Por isso, não é feita qualquer referência a outros tipos de linguagens como sejam as relacionadas com inteligência artificial (por exemplo, Prolog) ou com processamento simbólico (por exemplo, Lisp).

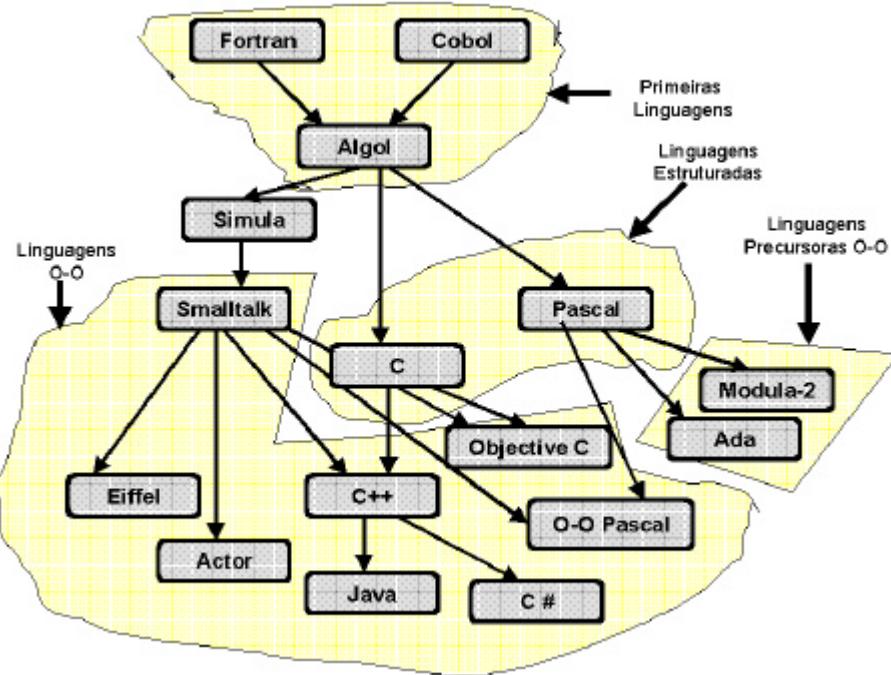



Figura 3.3: A evolução das linguagens de programação.

Se ao nível das linguagens a evolução ao longo do tempo tem sido significativa, o mesmo aconteceu em termos de outros conceitos da engenharia de software, o que vamos analisar de seguida.

3.3 O Desenvolvimento Ad-Hoc

Em face das potencialidades limitadas disponibilizadas ao nível das linguagens de programação, e dos restantes componentes de suporte tecnológico (computadores, sistemas operativos, tecnologias de armazenamento de informação), as primeiras aplicações foram construídas sem a utilização de uma metodologia de desenvolvimento formal, correspondendo ao que normalmente se designa por "programar e corrigir" (do inglês *"build and fix"*). A atenção estava concentrada na programação e na resolução das diversas restrições de natureza técnica, nomeadamente nas capacidades limitadas do hardware da altura. As profissões por excelência na área de informática eram a do programa-

dor que desenvolvia e testava os programas e a do operador que os executava.

Neste período, muitas organizações implementaram sistemas em ambientes proprietários, tipo *mainframe,* utilizando a linguagem Cobol para programar, e ficheiros para o armazenamento da informação; estes desenvolvimentos eram caracterizados por:

- Envolvimento limitado dos utilizadores.
- Identificação de requisitos efectuada de forma inadequada.
- Fraca utilização de técnicas de análise e desenho, que quando muito tinham uma perspectiva ad-hoc e eram de natureza empírica.
- Inexistência de ferramentas de apoio a todo o processo.
- Tarefas de desenvolvimento muito demoradas.
- Sistemas de acesso à informação pouco flexíveis.

Esta preocupação tecnológica implicou que frequentemente os requisitos dos utilizadores fossem relegados para segundo plano, o que na altura não constituía um problema demasiado grave, uma vez que as aplicações eram simples, limitando-se a substituir (de forma limitada) tarefas realizadas manualmente por processos automáticos. À medida que a complexidade aumentou, tal situação tornou-se cada vez mais problemática, o que foi ainda agravado por outras situações:

- Os informáticos eram reconhecidamente bons programadores, mas raramente tinham a preocupação de compreender o negócio e de utilizar um discurso e expressões aceites e compreendidas por pessoas não técnicas. As relações interpessoais não eram as melhores e ainda hoje vivemos as consequências desses primeiros tempos.
- Não eram aplicados técnicas ou mecanismos de controlo e de gestão de projecto, e por isso estes ultrapassavam frequentemente custos e prazos estimados.
- A fraca qualidade do produto final implicava correcções frequentes, o que justificava o significativo peso relativo das tarefas de manutenção e a consequente diminuição do tempo disponível para o desenvolvimento de novas soluções.

Apesar destes problemas todos, o ritmo de adopção de sistemas informáticos foi crescendo, o que só veio reforçar a necessidade de re-

ver a abordagem de desenvolvimento seguida de modo a introduzir e aplicar um processo sistematizado e formalizado de desenvolvimento.

3.4 As Metodologias Estruturadas

Durante os anos 70 e 80 assistiu-se a um importante salto qualitativo com a introdução de um conjunto de metodologias que se baseavam essencialmente em técnicas estruturadas de decomposição funcional. Estas metodologias tinham por objectivo formalizar o processo de identificação de requisitos, de modo a reduzir as possibilidades de má interpretação dos mesmos e introduzir técnicas baseadas nas melhores práticas ao processo de análise e desenho. A designação de **metodologias estruturadas** deriva da aplicação de um conjunto de princípios semelhante ao utilizado pelas linguagens de programação com o mêsmo nome, nomeadamente o principio da decomposição funcional.

3.4.1 Contexto e Motivação

Foi neste contexto que apareceram pela primeira vez os conceito ciclo de vida e de metodologias de desenvolvimento de software, com uma sequência de fases e actividades, com *inputs* e *outputs*, regras, intervenientes, técnicas, notações, ferramentas, documentação, técnicas de gestão, etc, com o objectivo de prestar mais atenção ao processo global, e menos à programação. A maioria destas metodologias adoptaram o modelo em cascata, apresentado na Secção 2.6 deste livro, em que cada actividade tem que ser completada e finalizada antes que a actividade seguinte possa ser iniciada.

As metodologias estruturadas tradicionais estavam essencialmente orientadas segundo uma de duas abordagens: (1) uma mais orientada para a decomposição funcional do sistema e a identificação dos respectivos processos; (2) outra mais centrada nos conceitos e nos dados das organizações. Estas duas perspectivas são normalmente designadas por análise funcional e análise orgânica.

Na **análise funcional**, funções, algoritmos e actividades são a preocupação central do engenheiro de software, incidindo toda a análise nas funcionalidades que o sistema deve realizar e numa fase posterior na definição de como essas funcionalidades serão implementadas. Esta abordagem era facilmente concretizada (ou mesmo condicionada) pela estrutura que as linguagens de programação tradicionais ofereciam, com base na definição de blocos de código que permitiam algum nível de autonomia e de encapsulamento de informação.

No entanto, a possibilidade dos dados serem globalmente visíveis e manipulados por todo o programa (o que era então permitido pelas linguagens de programação), reduzia significativamente essas características. Esta estrutura era pouco flexível, no sentido da incorporação de novas funcionalidades e da correcção de erros de implementação, uma vez que alterações num determinado ponto do programa poderiam produzir consequências importantes e não facilmente identificáveis noutros pontos do programa.

Na **análise orgânica**, a atenção concentra-se nos dados, colocando os conceitos e as entidades manipuladas no negócio como os elementos mais importantes do desenvolvimento do sistema. Esta preocupação incide sobretudo na compreensão e no agrupamento de dados comuns, e na identificação de relações entre esses grupos de informação (veja-se por exemplo a Figura 3.9 e a discussão da Secção 3.7.1 relativamente a este assunto). A ideia é que mesmo quando uma aplicação muda, os conceitos principais devem permanecer e continuar a ser utilizados, sem alterações significativas. A análise de dados envolve a recolha, validação e classificação das entidades, atributos e relações existentes num determinado domínio do problema.

3.4.2 Conceitos Básicos

As metodologias estruturadas introduziram um conjunto de conceitos, a maioria dos quais concretizados em diversos diagramas, dos quais destacamos os seguintes:

- O conceito **Processo (de negócio)** é uma sequência de actividades, que processam vários *inputs* e produzem vários *outputs*. Cada processo pode ser subdividido em processos mais elementares até se atingir um nível que não é possível decompor mais. É uma definição muito semelhante à de processo de desenvolvimento de software, como seria de esperar.

- **Fluxo de Informação** representa toda a circulação (e o respectivo suporte) de informação que ocorre numa organização de forma a executar os processos de negócio.

- **Repositório de dados** representam os conceitos sobre os quais importa à organização reter informação para utilização futura.
- **Entidade** representa todos os conceitos de negócio, entidades físicas ou abstractas, internas ou externas à organização, e que são relevantes para o desempenho adequado da função da organização.
- **Estado de uma entidade** é representado por um conjunto de valores que os atributos da entidade podem assumir.
- **Evento** é um acontecimento que é desencadeado internamente ou externamente ao sistema, ou pode representar simplesmente a passagem de tempo, e que despoleta uma mudança de estado.

Os três primeiros conceitos são exclusivos das abordagens orientadas a processos, enquanto os três últimos (mas sobretudo o quarto) são comuns aos métodos orientados a processos e aos dados.

É relativamente fácil a identificação destes conceitos numa situação concreta, o problema é conseguir obter uma descrição (isto é, um enunciado) bem organizada e objectiva dessa situação. Apresenta-se de seguida um enunciado que descreve o funcionamento da gestão de compras numa empresa, e nele são identificados os principais processos que ocorrem (e que se encontram sublinhados). Este exemplo será utilizado na Secção 3.4.3 para se clarificar as notações mais frequentes nestas metodologias.

> Pensemos no caso da gestão de compras de materiais de uma empresa. Sempre que algum funcionário tem necessidade de comprar bens para as suas actividades, este preenche uma requisição, onde identifica os bens em questão (com base na consulta de uma lista anteriormente disponibilizada a todos os funcionários), a qual envia para o seu director. Este, depois de analisar o pedido, pode ou não autorizar o mesmo. Neste segundo caso, o requisitante recebe uma notificação, e o processo pára. No caso do pedido ser aprovado, o requisitante preenche uma encomenda que envia para o fornecedor por ele seleccionado.
>
> A entrada dos bens ocorre sempre no armazém, onde é conferido o material recebido com a guia de remessa que o acompanha, bem como a(s) encomenda(s) que lhe deram origem (uma encomenda pode ser satisfeita de várias vezes). Após a última entrega, que completa a encomenda, a empresa confere a factura que entretanto lhe foi sido enviada, relaciona-a com as encomendas respectivas, e se tudo estiver correcto, é emitido um meio de pagamento ao fornecedor (poderá ser em cheque ou por transferência bancária).

No enunciado anterior encontram-se destacados os processos de negócio referidos. Este conceito de processo de negócio é, normalmente, o mais relevante no caso das metodologias estruturadas: os processos são os primeiros a ser identificados e conduzem toda a análise e especificação posterior.

3.4.3 Técnicas e Notações mais Utilizadas

As metodologias estruturadas propuseram diversos tipos de notações, sendo de destacar as seguintes representações gráficas e não-gráficas:

- **Diagramas de fluxo de dados** (*data flow diagram*): são especificações orientadas aos processos, em que se identificam graficamente processos, entidades externas, fluxos de informação e repositórios de dados. São os principais diagramas utilizados na modelação de processos. Estes diagramas podem ser construídos em vários níveis, uma vez que existe a possibilidade de especificar um processo através de um diagrama de maior detalhe. O diagrama de fluxo de dados de primeiro nível é nalgumas metodologias designado por diagrama de contexto.

- **Diagramas entidade associação** (*entity relarionship diagram*): são especificações gráficas que representam as relações estáticas de um sistema, designadamente as entidades, com os seus atributos, e a forma como estas se encontram associadas.

- **Diagrama de transição de estados** (*state transition diagram*): é uma representação gráfica dos estados em que um sistema ou uma entidade se pode encontrar ao longo da sua existência, e dos eventos que desencadeiam as transições entre estados.

- **Diagrama do ciclo de vida de entidade** (*entity life cycle*): consiste numa versão adaptada de um diagrama de estados, em que o objectivo é descrever a evolução de uma entidade ao longo da sua existência, designadamente as operações de criação, alterações significativas e eliminação do sistema.

- **Dicionários de dados**: são repositórios de definições de todos os elementos e conceitos utilizados e manipulados pela organização e respectivos sistemas de informação e que incluem entre outros os dados, ficheiros, processos e entidades.

- **Matrizes entidade-processo**: são matrizes que demonstram as relações existentes entre as entidades e os processos, isto é, permitem identificar que entidades intervêm em que processos.

- **Fluxogramas**: diagramas que expressam os passos de execução de algoritmos e processamentos realizados no sistema.

É ainda de referir a técnica da **Normalização**, que consiste num processo de construção do esquema de uma base de dados a partir da lista de entidades e dos respectivos atributos, aplicando um conjunto de regras designadas por formas normais do modelo relacional, cuja preocupação é eliminar redundâncias na estrutura de dados e garantir a integridade da informação. Para mais detalhe aconselhamos o leitor interessado a consultar [Silberschatz98].

O elevado número de metodologias disponíveis no mercado fez com que a simbologia utilizada variasse normalmente conforme a metodologia adoptada. Por exemplo, na Figura 3.4 podemos observar como o mesmo conceito "processo" é representado por símbolos gráficos distintos, conforme as metodologias de Yourdon e SSADM.

Figura 3.4: "Processo" representado diferentemente segundo as notações de Yourdon e SSADM.

A Figura 3.5 ilustra como o enunciado da gestão de compras pode ser representado através de um diagrama de fluxo de dados segundo a notação de Yourdon. Nele podemos observar os processos (representados por ovais), as entidades intervenientes (rectângulos), os repositórios de dados (rectângulos abertos lateralmente) e os fluxos de informação (setas).

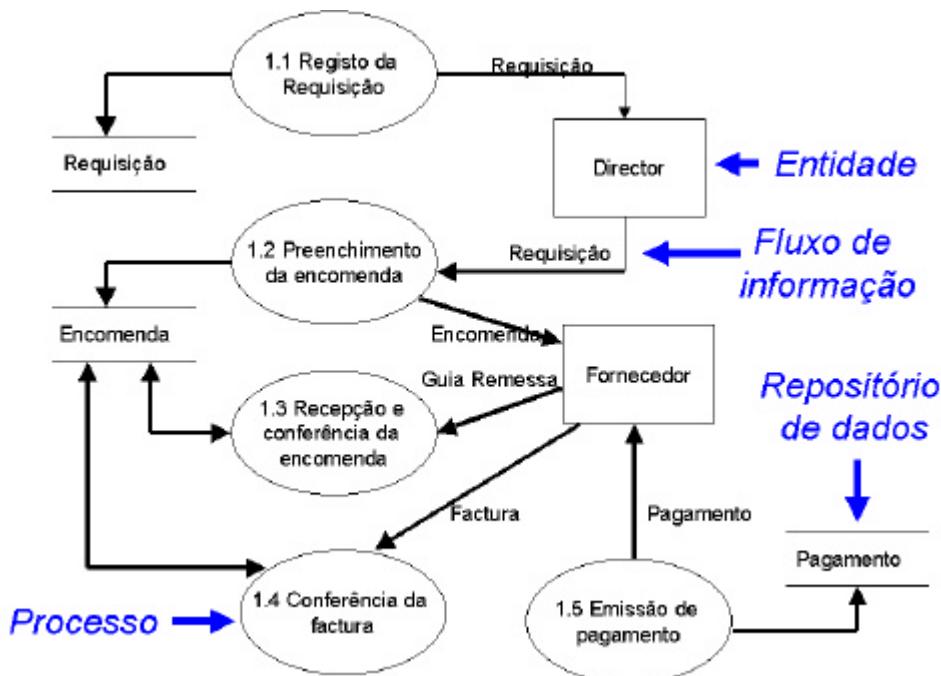



Figura 3.5: Diagrama de fluxo de dados do problema de gestão de compras.

Na figura anterior, poderíamos representar alguns processos com um nível de detalhe superior, e para isso já dispomos de informação no enunciado. Por exemplo, no caso do processo registo da requisição (1.1), o respectivo detalhe poderia incluir, e por esta ordem, a detecção da necessidade, a consulta à lista de bens disponíveis, o preenchimento da requisição, e o envio para o director. Esta informação seria também adequadamente representada por um diagrama de fluxo de dados, de nível mais detalhado.

Outro tipo de diagrama muito utilizado na literatura é o diagrama entidade associação (*entity relationship diagram*), que apresenta a visão estática do sistema, identificando as entidades sobre as quais interessa guardar informação, bem como o respectivo relacionamento. Na Figura 3.6 apresentamos o que poderia ser um diagrama deste tipo para o caso do problema da gestão de compras.

Qualquer relação expressa num diagrama entidade associação pode ser sempre analisada na perspectiva de ambas as entidades intervenientes, e implica a sua caracterização segundo dois conceitos adicionais: a cardinalidade (número máximo de ocorrências de cada uma na relação) e modularidade (número mínimo de ocorrências de cada uma, o que nos permite identificar quais são obrigatórias ou opcionais). Por exemplo, na relação entre "requisições" e "encomendas" do exemplo anterior, a cardinalidade desta relação, representada pelos símbolos mais próximos de cada rectângulo, lê-se "uma requisição pode ser satisfeita por várias encomendas"; a modularidade pode ser lida "podemos ter uma requisição que não dá origem a nenhuma encomenda" (é o caso em que o director não aprova o pedido).

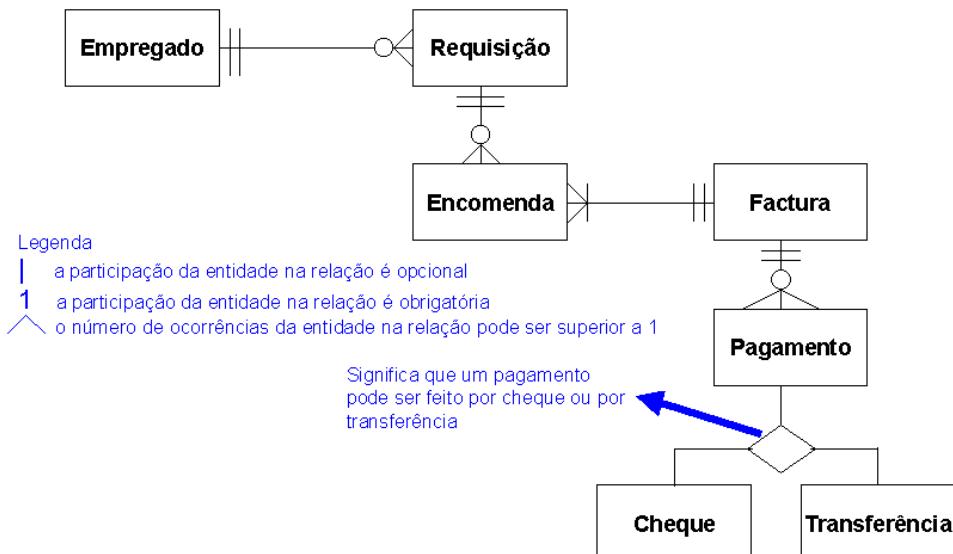



Figura 3.6: Diagrama entidade associação para o exemplo da gestão de compras.

Estes diagramas melhoraram o rigor da especificação, uma vez que foram definidas algumas regras às quais é necessário obedecer na sua elaboração. Adicionalmente, a representação gráfica dos conceitos facilitou a sua compreensão por todos os intervenientes no processo (utilizadores e informáticos). No entanto, continuam a apresentar algum grau de subjectividade, até porque normalmente prevêem a utilização de descrições em linguagem natural para complementar a modelação efectuada. Além disto, a utilização de alguns termos e símbolos já muito próximos da tecnologia dificultou por vezes a compreensão dos diagramas por parte de participantes não técnicos.

Os diagramas até agora apresentados incluem-se no grupo das notações semi-formais, uma vez que tinham um conjunto de regras associadas, e que deviam ser aplicadas na sua elaboração. No entanto, não era possível através destes diagramas garantir e demonstrar a respectiva correcção, face às funcionalidades identificadas. No sentido de resolver este tipo de problema, foram propostas notações formais, que se caracterizam por adoptar conceitos muito próximos da matemática, com o rigor e formalismo correspondentes, sendo deste modo possível demonstrar a correcção da especificação, o que é relevante

num número restrito, mas importante, de domínios de aplicação como seja a medicina, indústria militar ou a aeronáutica. Algumas notações mais conhecidas são:

- **Redes de Petri**: diagramas particularmente adequados para a representação de sistemas com problemas de concorrência e com restrições a nível de sincronização; utiliza conceitos como transições, funções de input e de output [Leveson87].
- **Linguagem Z**: linguagem de especificação formal, com simbologia e conceitos matemáticos e lógica de primeira ordem (conjuntos, tipos de dados, constantes, definições de estado, estado inicial, operações) [Spivey88].

Sai fora do âmbito deste livro aprofundar a aplicação das técnicas referidas. Para o leitor mais interessado aconselhamos a leitura da bibliografia recomendada.

3.4.4 Principais Metodologias

Foi referido no Capítulo 2 que o número de metodologias propostas para o desenvolvimento de software atingiu um número demasiado elevado, o que torna virtualmente impossível a sua apresentação em qualquer livro. Por isso, enumeramos de seguida algumas das metodologias estruturadas conhecidas que maior relevância e divulgação tiveram. Para além destas, existiram outros contributos importantes que não incluímos aqui por não apresentarem uma perspectiva integrada de todo o processo de desenvolvimento, mas apenas sugerirem notações ou técnicas de modelação. É o caso, por exemplo, das propostas de Tom DeMarco [DeMarco78] e de Meiler Page-Jones [Page-Jones80].

SSADM

A metodologia mais divulgada e que alcançou maior sucesso foi o SSADM (*Structured Systems Analysis and Design Methodology*), proposta em 1981 por Learmonth, e alvo de sucessivas revisões, a última das quais em meados da década de 90, com o aparecimento da versão 4+ [Weaver98]. Durante muito tempo (e ainda hoje) foi conside-

rada a metodologia de referência e ensinada em diversos cursos universitários. No Reino Unido, foi durante muito tempo obrigatória a sua utilização em todos os projectos relacionados com o desenvolvimento de sistemas de informação a nível governamental.

O SSADM propõe a modelação de um sistema segundo três perspectivas complementares: (1) a sua funcionalidade; (2) a sua estrutura; e (3) a sua evolução ao longo do tempo. A primeira é representada através de diagramas de fluxo de dados (DFD), a segunda é obtida através de diagramas de entidade associação (DEA) e a terceira pelos diagramas de ciclos de vida de entidades. Estas três visões são integradas; por exemplo, os DFD são comparados com os DEA, de forma a garantir que cada entidade referida num DEA é criada por algum processo especificado num DFD.

É uma metodologia concebida sobretudo para a análise e desenho do sistema, não contemplando as tarefas relacionadas com a implementação, testes e instalação do mesmo. Integra diversas notações orientadas quer para a modelação dos processos quer dos dados. A sequência de actividades envolve:

- A realização de um estudo de viabilidade, de modo a avaliar até que ponto o sistema tem custos aceitáveis.
- A análise de requisitos do negócio.
- A especificação dos mesmos requisitos.
- A especificação lógica do sistema, de modo a determinar a forma como os requisitos identificados são implementados.
- O desenho físico do sistema.

Para o leitor mais interessado aconselhamos uma leitura da referência anteriormente indicada.

STRADIS

Stradis (*Structured analysis, design and implementation of information systems*) foi uma metodologia desenvolvida por Gane e Sarson em finais da década de 70 [Gane82], baseada na filosofia de decomposição funcional *top-down*, e suportada essencialmente pela utilização de diagramas de fluxos de dados.

Yourdon Systems Method

Yourdon Systems Method, é uma metodologia proposta por Edward Yourdon, revista em 1993 e descrita com algumas actualizações em [Yourdon99]; é semelhante à Stradis pois recorre muito à decomposição funcional, mas também atribui uma importância significativa à estrutura dos dados.

Engenharia de Informação

A "Engenharia de Informação", proposta por James Martin em 1989 [Martin89], integra muitos dos conceitos, melhores práticas, modelos e técnicas das metodologias estruturadas e do desenvolvimento de software dos anos 80 numa abordagem coerente a todas as actividades do processo de desenvolvimento de software. As suas raízes estão no trabalho originalmente desenvolvido pela IBM na sua metodologia *Business Systems Planning*.

Ao contrário das outras, a Engenharia da Informação tem uma preocupação estratégica com a definição dos sistemas de informação, o que é expresso nos diferentes estágios de evolução do processo designadamente: (1) planeamento estratégico dos sistemas de informação; (2) análise de áreas do negócio; (3) desenho de sistemas e (4) implementação.

3.5 Metodologias Orientadas por Objectos

As técnicas e metodologias estruturadas descritas na secção anterior apresentam vários problemas, entre os quais podemos destacar:

- Não conseguem lidar adequadamente com o problema da complexidade e do tamanho crescente dos sistemas.
- Não resolvem o problema da crescente actividade de manutenção do software.
- Verifica-se com frequência a má compreensão dos requisitos do utilizador, por parte dos intervenientes técnicos.
- Permanece a dificuldade de lidar com alterações aos requisitos.
- A integração e reutilização de módulos e componentes do sistema não são fáceis.
- Os erros de concepção são descobertos tardiamente.
- A qualidade do software é baixa e o seu desempenho inadequado.
- Não é fácil identificar quem fez o quê, quando e porquê.

A aplicação de diversas das melhores práticas actuais de engenharia de software (ver Secção 2.4) veio solucionar algumas destas questões e esteve na origem do conceito da orientação por objectos. No entanto, é importante reforçar a ideia de que em muitos casos essas melhores práticas podem ser seguidas independentemente de se utilizarem métodos estruturados ou orientados por objectos.

A orientação por objectos, se bem que em termos práticos tenha sido primeiramente concretizada ao nível das linguagens de programação, não tem impacto apenas a esse nível. O conceito da **orientação por objectos** (OO ou O-O, do inglês *object-oriented*) baseia-se de facto numa nova forma de analisar o mundo. A abordagem seguida "reproduz" a forma como o ser humano se apercebe e expressa a realidade que o rodeia. Ele classifica e subdivide o mundo em diferentes objectos, com base nas diferenças e semelhanças existentes ao nível das características e comportamento dos mesmos objectos. As técnicas orientadas por objectos identificam e definem cada objecto de modo a reutilizá-lo, da mesma forma que o ser humano acumula conhecimento com base no previamente adquirido.

Nas abordagens orientadas por objectos, a perspectiva de modelação dos sistemas muda, uma vez que o mesmo conceito base é utilizado ao longo de todas as fases do processo, promovendo a reutilização e o encapsulamento da informação, e facilitando a manutenção.

As metodologias orientadas por objectos são por isso encaradas como uma das mais recentes panaceias (ou *silver bullets*, segundo Brooks [Brooks86]) para resolver os problemas do desenvolvimento de software, uma vez que são abordagens mais naturais que as baseadas em processos e dados, e os conceitos básicos são simples e reproduzem o mundo real.

A definição exacta do que se entende por orientação por objectos tem motivado largas discussões ao longo das últimas duas décadas, e recomendamos a leitura de algumas referências ([Berard93], [Taylor90], [Stroustrup88] e [Booch86]) para um melhor esclarecimento sobre este assunto. No entanto, pensamos que uma das visões mais simplificadas até hoje apresentada, e que expressa adequadamente o conceito foi elaborada por Coad e Yourdon em 1991 [Coad91], e indicava que o paradigma da orientação por objectos resultava da convergência de quatro outros conceitos, que serão definidos em secções mais à frente: objectos, classificação, herança e comunicação por mensagens.

3.5.1 Contexto e Motivação

As raízes da engenharia de software orientada por objectos podem ser encontradas no trabalho inicialmente desenvolvido na linguagem Simula [BirtWhistle79] em finais dos anos 60, que como o nome indica, estava particularmente vocacionada para a implementação de sistemas de simulação.

Desde o início dos anos 70 que três ideias independentes ganharam importância, com o objectivo de facilitar todo o processo de desenvolvimento de software, e que em última análise estão na base da estrutura de conceitos do paradigma da orientação por objectos. Estamos a referir-nos aos conceitos de encapsulamento de informação, de reutilização de código e da visão do mundo (e posterior modelação) segundo um conjunto de objectos, e não segundo uma perspectiva

funcional. Estes conceitos estão na base da primeira linguagem classificada como verdadeiramente suportando este paradigma, a linguagem Smalltalk [Goldberg89], concebida nos laboratórios PARC da Xerox. No entanto, e até meados da década de 80, a maioria das iniciativas relacionadas com o paradigma da orientação por objectos situava-se ao nível da programação.

Tal como descrito no seu livro "*Object-Oriented Analysis*" [Coad91], Peter Coad e Edward Yourdon identificam as motivações principais para a realização de actividades de análise segundo o paradigma da orientação por objectos:

- Poder tratar domínios de problemas mais complexos.
- Melhorar a interacção entre o analista e o especialista do problema.
- Aumentar a consistência interna dos resultados da análise.
- Representar explicitamente aspectos comuns a diversos conceitos.
- Construir especificações capazes de resistir à mudança.
- Reutilizar resultados da análise.
- Conceber uma representação consistente para a análise e desenho.

3.5.2 Conceitos Básicos

As abordagens orientadas por objectos são muito ricas em termos da terminologia utilizada. Podemos distinguir essencialmente dois grandes grupos de conceitos: (1) os princípios, que constituem um conjunto de ideias base de todo o paradigma, e alguns deles são mesmo os requisitos necessários para um sistema ser considerado orientado por objectos; (2) e as restantes noções base, comuns a todos os sistemas orientados por objectos.

Princípios Orientadores do Paradigma

No grupo dos princípios orientadores e que definem o paradigma podemos incluir os seguintes conceitos:

Conceito

- **Encapsulamento da informação** é o processo de "esconder" todos os detalhes de um objecto que não contribuem para as suas características essenciais nem para a disponibilização de funcionalidades para o seu exterior. Pode ainda ser encarado como a localização de funcionalidades numa única abstracção auto-contida, que esconde a respectiva implementação e decisões de desenho, através da disponibilização de uma interface pública. O conjunto de operações que são acessíveis do exterior constitui a interface do objecto. Esta característica permite a criação de objectos estáveis e reutilizáveis, reproduzindo o mundo real de forma correcta.

Conceito

- **Herança** representa a definição de relações entre classes através da qual uma subclasse partilha, acrescenta ou redefine operações e atributos a partir de uma ou mais superclasses; uma subclasse é uma especialização de uma ou mais superclasses. É uma forma de aumentar a reutilização do que é comum, permitindo adicionalmente acrescentar detalhes específicos. Este termo aparece associado às noções de generalização e de especialização.

Conceito

- **Polimorfismo** é a capacidade de "esconder" várias implementações distintas através de uma única interface. Outra forma de definir esta propriedade é dizer que ela representa a capacidade de objectos diferentes responderem de forma diferente à mesma mensagem. Tal é concretizado pelo facto de uma operação aceitar argumentos de tipos diferentes ou mesmo desconhecidos. É utilizada em situações em que uma operação é partilhada na maior parte dos casos, mas em que pelo menos uma subclasse necessita de uma versão específica.

Conceito

- **Abstracção** é a representação concisa duma ideia ou objecto mais complexa, incidindo sobre as características essenciais do objecto. As abordagens tradicionais concretizam esta ideia pelas abstracções funcionais (processos), enquanto os métodos orientados por objectos utilizam objectos.

Os quatro princípios anteriores são considerados necessários e suficientes para considerar uma linguagem como sendo orientada por objectos; aquelas que não suportam as noções de herança e/ou polimorfismo são normalmente designadas por baseadas em objectos. Para além destes, existem ainda outras noções importantes:

- **Modularidade** é a decomposição lógica e física de conceitos em unidades mais elementares, de forma a facilitar a aplicação dos princípios da engenharia de software.

- **Concorrência** é a propriedade que distingue um objecto activo de outro não activo.
- **Persistência** é a propriedade de um objecto através da qual a sua existência perdura no tempo e/ou no espaço.

Outros Conceitos Chave do Paradigma da Orientação por Objectos

Para além dos princípios que funcionam como as grandes linhas orientadoras do paradigma da orientação por objectos, existem outros conceitos que são fundamentais para a boa compreensão do mesmo, nomeadamente os conceitos de objectos e classes, para além de outros que estão directamente relacionados com estes.

O conceito básico é obviamente o de **objecto**, para o qual não existe uma definição única. Pelo contrário, praticamente cada metodologia e respectivo autor apresentam definições próprias, como se pode observar de seguida:

- Firesmith [Firesmith96]: um objecto pode ser definido como uma abstracção de software que modela todos os aspectos relevantes de uma única entidade conceptual ou tangível, que pertence ao domínio da solução.
- *Object Management Group*: um objecto é uma coisa, criada como uma instância de um tipo de objectos. Cada objecto tem uma identidade única distinta e independente de quaisquer das suas características. Cada objecto tem uma ou mais operações.
- Berard [Berard93]: objectos são as entidades reais ou conceptuais que podem ser encontradas no mundo que nos rodeia.
- Booch [Booch94]: algo ao qual se pode fazer qualquer coisa; tem estado, comportamento e identidade.

- Coad e Yourdon [Coad91]: uma abstracção de qualquer coisa no domínio de um problema, reflectindo a capacidade do sistema de manter informação sobre ele e de interagir com ele; é um encapsulamento de valores de atributos e dos seus serviços exclusivos.
- Rumbaugh [Rumbaugh91]: um objecto é um conceito, abstracção ou coisa com fronteiras bem definidas e com significado para o problema em questão; promove a reutilização e funciona como uma base concreta para a respectiva implementação em software.
- Shlaer e Mellor [Shlaer88]: um objecto é uma abstracção de um conjunto de coisas do mundo real de tal forma que todos os elementos do conjunto (instâncias) têm as mesmas características e respeitam as mesmas regras e políticas.

Resumidamente, e de forma a integrar estas diversas definições, podemos dizer que os objectos representam entidades físicas, conceptuais ou apenas necessárias para a representação em computador (por exemplo, estruturas de dados); um objecto pode ser um conceito, uma abstracção ou uma entidade, com fronteiras bem definidas e que tem um significado para um problema e respectiva solução; um objecto tem estado, comportamento e identidade.

Alguns autores enumeram as fontes onde pesquisar os possíveis candidatos a objectos, independentemente do problema a solucionar:

- Shlaer e Mellor: entidades e coisas tangíveis, funções de pessoas, organizações, incidentes, eventos, interacções, especificações.
- Coad e Yourdon: estruturas, outros sistemas, dispositivos, coisas ou acontecimentos memorizados, funções desempenhadas, procedimentos operacionais, locais geográficos, unidades organizacionais.

Os **atributos** são propriedades nomeadas de um objecto que indicam os valores possíveis que esse atributo pode assumir ao longo do tempo; o estado de um objecto é definido pelo valor dos seus atributos.

O **comportamento** de um objecto é definido como o conjunto de acções que um objecto pode realizar de forma independente. Normalmente existem dois termos para referir os mecanismos utilizados para implementar este conceito: ao nível da análise o termos mais utilizado é o de operação ou serviço, enquanto o termo método se

refere normalmente à implementação de uma operação ao nível da programação. A maioria das operações realizadas responde a perguntas ou alteram o estado do objecto. Na prática, implementam um serviço e/ou funcionalidade que pode ser requerido por qualquer objecto que faça parte do universo do problema.

Para melhor compreendermos os conceitos de estado e de comportamento, utilizemos o exemplo de uma conta bancária. Neste caso, o seu estado é determinado pelos valores assumidos pelos atributos que são necessários para a caracterizar - titular, saldo - enquanto que o seu comportamento é determinado pelos serviços que disponibiliza para outros objectos interagirem com ela, nomeadamente depositar dinheiro, levantar dinheiro e consultar o saldo da conta.

As **classes** são descrições de grupos de objectos com propriedades (atributos), comportamento (operações) e relações comuns. É uma das concretizações do conceito abstracção no paradigma da orientação por objectos. Uma classe pode ser vista como *template* para um determinado objecto e todos os que lhe forem semelhantes, ou como uma fábrica, que produz tantos objectos idênticos quanto necessário.

Em geral, os atributos permitem identificar a classe, definir as características especiais da classe, definir o estado da classe, reter alguma informação sobre a classe e identificar o comportamento do objecto. Para especificar detalhadamente um atributo deve-se identificar o domínio dos seus valores, as unidades de medida, o valor por omissão, o valor inicial, as condições de leitura e escrita e eventuais condições relacionadas com outros atributos do objecto ou da classe.

Dado um enunciado de um problema, a análise orientada por objectos procura identificar os objectos concretos e as respectivas classes, a partir de todos os substantivos que dele sejam extraídos. O seu objectivo último será sempre identificar:

- Um conjunto de classes que representem totalmente o domínio do problema.
- Os atributos de cada classe.
- A interface e os serviços de cada classe.
- As diversas relações entre classes.
- Objectos concretos que seja necessário particularizar.

Cada metodologia propõe diferentes formas de chegar a esta informação, e normalmente referem alguns critérios. Por exemplo, [Coad91] indica um conjunto de questões a colocar, de modo a averiguar se um substantivo do enunciado constitui um objecto ou classe:

- A informação sobre o objecto tem que ser retida de modo a que o sistema funcione adequadamente?
- O objecto realiza operações que alteram atributos de outros objectos?
- O objecto tem mais do que um atributo?
- Outros objectos aparentemente idênticos disponibilizam operações idênticas?
- O objecto produz ou consome informação essencial para o funcionamento do sistema?

Noutras situações, são indicadas *checklists* de circunstâncias em que um potencial objecto não o deve ser, por exemplo:

- Objectos ou classes com apenas um atributo.
- Classes com apenas um objecto.
- Objectos ou classes sem serviços aplicáveis, ou que não produzem um resultado visível para o problema em análise.
- Objectos ou classes que não sejam relevantes para a solução do problema.
- Objectos ou classes que de facto correspondem a atributos de outros objectos.

Se considerarmos novamente o enunciado do exemplo da gestão de compras, podemos verificar que a nossa pesquisa passa a incidir sobre outras palavras (substantivos em vez de verbos), devido à mudança de paradigma e aos conceitos base fundamentais que lhes estão subjacentes (objectos em vez de processos).

Pensemos no caso da gestão de compras de materiais de uma empresa. Sempre que algum funcionário tem necessidade de comprar bens para as suas actividades, este preenche uma requisição onde identifica os bens em questão (com base na consulta de uma lista anteriormente disponibilizada a todos os funcionários), a qual envia para o seu director. Este, depois de analisar o pedido, pode ou não autorizar o mesmo. Neste segundo caso, o requisitante recebe uma notificação, e o processo pára. No caso do pedido ser aprovado, o requisitante preenche uma encomenda que envia para o fornecedor por ele seleccionado.

A entrada dos bens ocorre sempre no armazém, onde é conferido o material recebido com a guia de remessa que o acompanha, bem como a(s) encomenda(s) que lhe deram origem (uma encomenda pode ser satisfeita de várias vezes). Após a última entrega, que completa a encomenda, a empresa confere a factura que entretanto lhe foi sido enviada, relaciona-a com as encomendas respectivas, e se tudo estiver correcto, é emitido um meio de pagamento ao fornecedor (poderá ser em cheque ou por transferência bancária).

As operações realizadas por objectos podem ser identificadas pela pesquisa no enunciado do problema de verbos associados a cada objecto. Essas operações podem ser agrupadas nas seguintes categorias:

- Modificadores: operação que altera o estado de um objecto.
- Selectores: operação que acede ao estado de um objecto.
- Iteradores: operação que permite que todas as partes de um objecto sejam acedidas segundo uma ordem bem definida.
- Construtor: cria um objecto e/ou inicializa o seu estado.
- Destrutor: liberta o estado de um objecto ou destrói-o.

Os objectos funcionam como caixas negras, isto porque "escondem" os detalhes da sua implementação aos objectos que a utilizam; o acesso aos objectos é efectuado através da interface por eles disponibilizada, composta por operações e por atributos. Os objectos comunicam entre

si por mensagens, que não são mais do que a invocação de um método com o mesmo nome, no contexto do objecto receptor da mensagem.

Figura 3.7: Modelo de comunicação entre objectos por mensagens.

Na prática, este modelo não é mais do que a invocação de uma função, tal como nas abordagens tradicionais; a diferença é que esta invocação é realizada no contexto de um objecto, o que significa que tem em conta o seu estado, traduzido nos valores que os seus atributos assumem. Por isso, o mesmo método executado com os mesmos parâmetros sobre objectos diferentes, mas ambos instâncias da mesma classe, pode produzir resultados diferentes, como se verifica no exemplo seguinte.

```
Classe classeA
{
  public:
    int x;
    int y;
    int soma() { return (x + y ) ;}
}
...
classeA a;
classeA b;
a.x = 10 ;
a.y = 20 ;
```

```
b. x = 5 ;
b.y = 8 ;

...
a.soma => obtem-se o valor 30
b.soma => obtem-se o valor 13
```

Este conceito de comunicação por mensagens pode ser visto de forma tão abstracta que mesmo a soma de dois números pode ser encarada como uma operação de envio de uma mensagem a um objecto. Por exemplo na operação **3 + 4**, 3 é o objecto receptor da mensagem, que é formada pelo selector + (que identifica o método a invocar no objecto receptor) e pelo objecto argumento 4. Uma mensagem é sempre formada por um selector e opcionalmente por um conjunto de parâmetros.

A **interface** é o conjunto de operações e atributos disponibilizados por uma classe, que consoante a visibilidade se pode dividir em três partes: pública (visível para todos os objectos do sistema); protegida (só visível pelas suas subclasses); privada (faz parte da interface mas não é visível para nenhuma outra classe do sistema, só está disponível na implementação da própria classe).

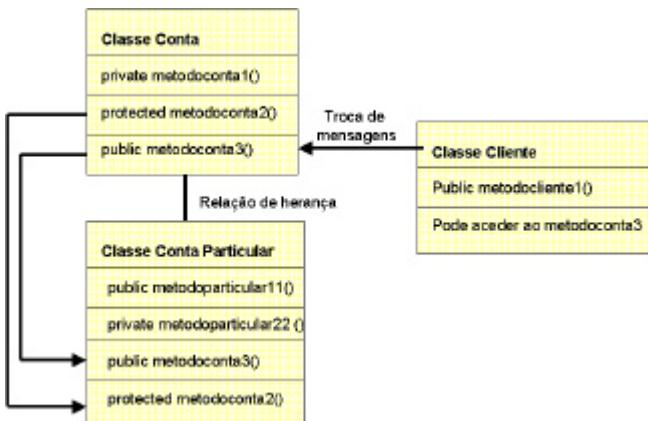



Figura 3.8: Visibilidade de métodos da interface.

Entre as diversas classes de um sistema podem ser estabelecidas diferentes tipos de relações:

- **Associação**: representam relações estruturais entre objectos de classes diferentes, e cuja informação que tem que ser preservada durante algum tempo; é expressa pelo verbo **ter**; podem-se sub-dividir em
- **Agregação**: é a conhecida relação entre o todo e as partes (*whole-part*); um exemplo possível é a relação "uma empresa tem empregados".
- **Composição**: forma de agregação em que a relação de pertença é forte e com tempos coincidentes; o objecto agregador é responsável pela criação e destruição do objecto que entra na sua composição; uma concretização desta relação é dizer que "o corpo humano tem uma perna".
- **Dependência**: relação em que uma mudança de estado num objecto (ocorrida pela recepção de uma mensagem) pode implicar o envio de uma mensagem a outro objecto; por exemplo, um automóvel, para andar, precisa de se abastecer na bomba de gasolina.
- **Generalização/Especialização**: relações entre classes que partilham a estrutura e comportamento; implementam o conceito de herança, que pode ser simples (uma classe tem apenas uma superclasse) ou múltipla (uma classe pode ter várias superclasses); esta relação é expressa pelo verbo **ser**, como no caso "um cão é um animal".

Algumas abordagens orientadas por objectos procuram definir conceitos que permitem agrupar classes; por exemplo, Wirfs-Brock considerou que um **sub-sistema** é um conjunto de classes (ou de outros sub-sistemas) que colaboram entre si para realizar um conjunto de responsabilidades. Não existem na prática, mas permitem a criação de entidades conceptuais úteis. Coad e Yourdon definem o conceito de assunto (*subject*) de forma idêntica. NO caso do UML, o utiliza-se o conceito de pacote com um fim idêntico. O principal objectivo deste tipo de conceitos é facilitar a compreensão da globalidade do sistema, ao agrupar outros conceitos relacionados.

3.5.3 Técnicas e Notações mais Utilizadas

A proliferação das metodologias que tinham como base os conceitos da orientação por objectos levou ao aparecimento de diversas notações e técnicas de modelação, que muitas vezes são partilhadas entre várias delas. Os recentes esforços de unificação permitiram que algumas dessas técnicas se tenham destacado.

As principais notações utilizadas estão praticamente todas presentes na metodologia da Rational, e fazem parte da linguagem de modelação UML, pelo que serão apresentadas detalhadamente nos capítulos que constituem a Parte 2 deste livro. Resumidamente, indicamos de seguida as principais notações a considerar:

- Diagrama de casos de utilização
- Diagrama de classes
- *CRC (Class-Relationship-Collaboiration) cards*
- Diagrama de pacotes
- Diagrama de estados
- Diagramas de interacção
- Diagrama de actividades
- Diagrama de eventos
- Diagrama de contexto
- Diagrama de fluxo de dados.

Independentemente dos nome dos diagramas utilizados por cada metodologia, cada uma apresenta notações para modelar a visão estática do sistema (a estrutura dos objectos, relações de agregação e de especialização, e a comunicação entre objectos) e outras notações para os conceitos dinâmicos (interacções, mudanças de estado, sequências de acções e mecanismos de temporização).

3.5.4 Principais Metodologias

Ao longo das décadas de 1980 e de 1990 surgiram inúmeras propostas de metodologias, sobretudo concentradas nas tarefas de análise e desenho, utilizando os conceitos relacionados com o paradigma da orientação por objectos. Sem pretender ser exaustivo, enumeramos de seguida algumas das metodologias mais significativas, quer pela sua utilização, quer pela relevância dos conceitos abordados.

Método de Booch

O método Booch foi proposto por Grady Booch em 1991 (cujo livro de referência teve uma segunda edição em 1994 [Booch94]), e baseia-se na ideia da repetição de actividades de um processo de desenvolvimento de modo a refinar o modelo em sucessivas iterações. As suas principais actividades estão orientadas para a identificação de classes e objectos e respectivas características e para a determinação das relações entre classes.

OMT (Object Modelling Technique)

O OMT foi proposto por James Rumbaugh em 1991 [Rumbaugh91], e concentrou as suas propostas na análise e desenho de software, às quais aplicou técnicas orientadas por objectos. A sua metodologia apresentava essencialmente três modelos principais: estático ou de objectos (onde se representavam classes, objectos, a hierarquia e outras relações), dinâmico (apresentava o comportamento dos objectos e do sistema global) e funcional (diagrama de fluxo de informação no sistema semelhante aos diagramas de fluxos de dados).

OOSE (Object Oriented Software Engineering)

O OOSE foi proposto por Ivar Jacobson em 1992 [Jacobson92], resulta da evolução do modelo *Objectory* (também do mesmo autor) e a sua maior contribuição foi a introdução da noção de caso de utilização que funciona como uma descrição da interacção entre o utilizador e o sistema.

OOAD (Object Oriented Analysis and Design)

O OOAD foi proposto por Peter Coad e Edward Yourdon em 1991 [Coad91], apresentava como uma das suas principais vantagens o facto de ser muito simples (ao nível dos conceitos, actividades e diagramas) o que o tornava um dos mais fáceis de compreender. As suas principais actividades relacioandas com a análise são no fundo aquilo que todos nós esperaríamos realizar num processo que aplicasse as noções da orientação por objectos:

- Identificar objectos utilizando critérios simples (substantivos).
- Definir uma estrutura de relações generalização-especificação.
- Definir uma estrutura de relações de associação (*whole-part)*.
- Identificar assuntos (subsistemas).
- Definir os atributos.
- Definir os serviços.

Método de Wirfs-Brock

A metodologia de Wirfs-Brock não efectua uma distinção clara entre análise e desenho, e a sua principal contribuição foi a definição de um diagrama designado por *CRC cards* (Class-Responsibility-Colaboration) que procura identificar as classes do sistema, a sua interface e as relações entre elas. As principais actividades consistem em avaliar a especificação do cliente; extrair classes candidatas por análise da especificação; identificar grupos de classes (superclasses); definir e atribuir responsabilidades para cada classe; identificar relações entre classes; identificar colaborações entre classes com base nas responsabilidades; e construir representações hierárquicas das classes

Para além destas metodologias, sem dúvida alguma que aquela que actualmente é mais relevante é a proposta pela Rational (e que na realidade se baseia em muitos dos conceitos aqui referidos) e que é apresentada em detalhe no capítulo 10. Outras que vale a pena mencionar são a de Shlaer e Mellor [Shlaer88], a de Edward Berard [Berard93] e a de James Martin [Martin92].

3.6 Outras Metodologias

Apesar dos benefícios que se reconhecem na utilização de metodologias (independentemente do paradigma utilizado), elas não estão isentas de críticas e de aspectos menos positivos:

- Complexidade nos conceitos, técnicas e aplicação.
- Desconhecimento global da metodologia e falta de competências dos informáticos para a sua execução com qualidade.
- Ferramentas que suportam a metodologia difíceis de utilizar.
- Constatação da ausência de melhorias significativas no processo e produto final.
- Concentração na análise da situação actual, menor importância aos objectivos futuros.
- Tempo que decorre até à disponibilização dos resultados finais.
- Rigidez na aplicação dos métodos e conceitos.

Como reacção a esta situação, um novo grupo de metodologias começou a aparecer nos últimos anos, que implicam um nível de formalismo muito menor. Muitas delas advogam a não realização de actividades de análise e desenho, e a produção de muito menos documentação por comparação com as metodologias estruturadas ou orientadas por objectos. Defendem que a principal documentação de um sistema é, ou deveria ser, o código fonte das aplicações desenvolvidas. Comungam entre si a ideia de que as principais actividades a realizar ao longo de todo o processo de desenvolvimento são essencialmente a programação e os testes.

Estes métodos são bastante adaptáveis, também como forma de responder adequadamente aquilo que eles consideram ser a imprevisibilidade dos requisitos ao longo do tempo. Concentram-se na satisfação das necessidades das pessoas (informáticos e utilizadores) e não na definição de processos. Partilham a ideia do desenvolvimento iterativo típico das abordagens orientadas por objectos, e reforçam a importância da actividade de testes.

Apesar delas partilharem um conjunto de ideias base comuns, não existe ainda uma designação formal para elas, a não ser o termo

lightweight (leve ou simplificado). São na sua maioria desconhecidas do grande público informático, apesar de algumas delas já estarem a ser aplicadas há vários anos. Algumas referências relevantes nesta área são as abordagens *XP - Extreme Programming* [Beck99], *Feature Driven Development* [Coad99] e *DSDM - Dynamic System Development Method*.

Tal como referido por Fowler [Fowler00], em determinadas circunstâncias estes métodos são particularmente aconselháveis, sobretudo se estivermos a falar de sistemas pequenos ou com requisitos incertos ou voláteis, em que os programadores são responsáveis, experientes e se encontram motivados e em que os clientes são igualmente participativos e responsáveis, ou ainda quando a equipa de desenvolvimento é relativamente reduzida e estável. Noutras situações será preferível um processo mais formal, nomeadamente sempre que o projecto exigir a alocação de equipas de maior dimensão, ou existir um contrato com âmbito e preços fixos, ou em que o risco seja elevado e o processo de controle do projecto deva ser reforçado.

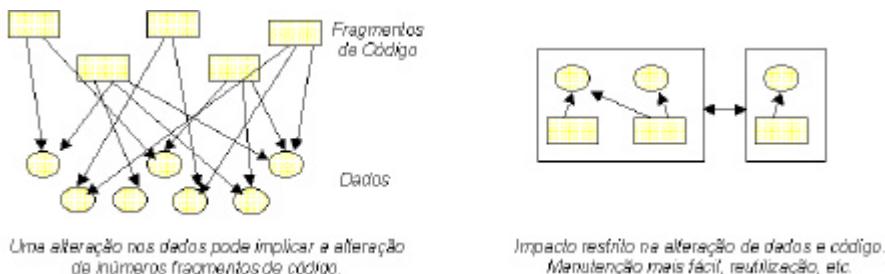
3.7 Comparação de Metodologias

A comparação das diversas metodologias actualmente existentes é uma tarefa complexa devido a um conjunto de dificuldades que se colocam e das quais podemos destacar as seguintes:

- Não existem metodologias iguais e portanto em qualquer comparação estaremos sempre a comparar conceitos por vezes não comparáveis.
- Muitas metodologias são influenciadas ou particularmente concebidas para serem utilizadas com linguagens de programação específicas.
- Muitas metodologias assumem um contexto de aplicação onde não existem os problemas que no mundo real têm que enfrentar.
- A abrangência das metodologias varia fortemente; algumas apenas descrevem um processo, outras incluem técnicas e notações.
- A comparação entre metodologias tem que considerar obrigatoriamente apenas um subconjunto das mesmas, e um subconjunto de funcionalidades.

- A própria definição do conceito metodologia pode ser limitativa.

Hoje em dia reconhece-se a vantagem clara do ponto de vista conceptual das abordagens orientadas por objectos face às estruturadas, pois apresentam:

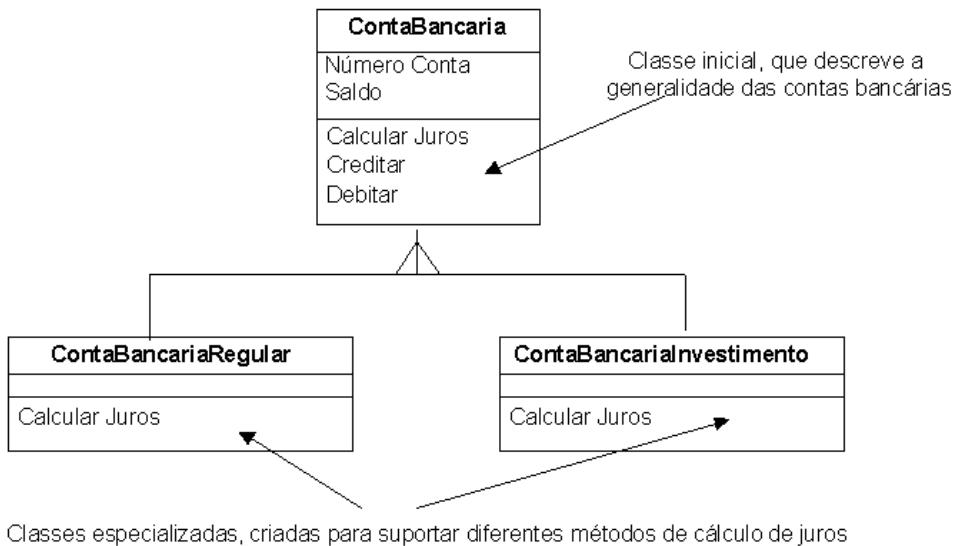
- Um único paradigma consistente ao longo de todo o processo, mais próximo do processo cognitivo humano.
- Facilitam a reutilização não apenas do código mas também da arquitectura global do sistema, o que potencia o aumento de produtividade dos informáticos.
- Apresentam modelos que reflectem mais adequadamente o mundo real.
- Não existe separação entre dados e processos, o conceito unificador agrega as duas visões.
- Os detalhes de implementação são escondidos do exterior pela aplicação de técnicas de encapsulamento da informação.
- A facilidade de realização das tarefas de manutenção é maior, em resultado das diversas características anteriormente enumeradas.
- O sistema construído é consequentemente mais estável.

Estas abordagens têm contudo os seus problemas, pois reconhece-se que nem sempre é fácil encontrar os objectos e classes apropriados no domínio do problema, uma vez que a maioria dos informáticos continua a pensar em termos funcionais. Para além disso, só recentemente começaram a surgir no mercado ferramentas de apoio ao processo de desenvolvimento segundo o paradigma da orientação por objectos.

Comparamos de seguida alguns aspectos concretos em como as abordagens orientadas por objectos se revelam mais adequadas que as estruturadas.

3.7.1 Gestão de Requisitos e Facilidade de Manutenção

A figura 3.9 ilustra a diferença entre os paradigmas do desenvolvimento estruturado versus o desenvolvimento orientado por objectos. No desenvolvimento estruturado, o sistema consiste num conjunto de dados que são usados (em leitura e/ou escrita) por inúmeras funções,

de forma mais ou menos independente. O sistema corresponde a uma malha arbitrária de inúmeras interdependências entre funções e dados. No desenvolvimento orientado por objectos, dados e funções são agregados conjuntamente numa entidade lógica (o objecto) que providencia uma interface bem definida para outros objectos comunicarem com ele. O sistema corresponde a uma malha de interdependências entre objectos.

Figura 3.9: Desenvolvimento estruturado versus desenvolvimento orientado por objectos.

É evidente que as abordagens orientadas por objectos facilitam significativamente a gestão das alterações de requisitos. Imagine-se as implicações que a alteração da estrutura dos dados implica em ambas as abordagens. Na abordagem estruturada, seria necessário adaptar consistentemente todas as funções que acedessem à estrutura de dados envolvida, e, na pior das situações, a todas as funções que dependessem funcionalmente dessas primeiras funções.

Por outro lado, nas abordagens orientadas por objectos, apenas seria necessário alterar a estrutura de dados que supostamente era definida no contexto interno de um determinado objecto. Caso essa alteração não implicasse a alteração da interface providenciada pelo objecto, não haveria mais nada a fazer!

3.7.2 Representação da Realidade

Nas abordageens orientada por objectos, as entidades do mundo real são captadas directamente por objectos: um carro é um objecto do tipo “carro”, uma conta bancária é um objecto do tipo “conta bancária”, etc.

Tal como as entidades do mundo real, os objectos apresentam dados e comportamento, bem como identidade própria.

Figura 3.10: Especialização de contas bancárias na abordagem OO.

Nas abordagens estruturadas, uma conta bancária pode ser representada através de um ou mais módulos com funções específicas que acedem a várias estruturas de dados de forma independente. Nestas abordagens, o foco é na definição do modelo de dados da conta bancária (isto é, no desenho da base de dados que suporta contas bancárias) ou, alternativamente, na definição dos processos/funções que acedem e manipulam contas bancárias. Consequentemente a esta abordagem, não é fácil explicitar especializações ou generalizações correspondentes a conceitos inerentes a entidades do mundo real que existam ou que venham a existir.

Por exemplo, não é trivial na abordagem estruturada introduzir-se posteriormente a noção de contas bancárias especializadas, o que é simples na abordagem orientada por objectos, tal como é representado na Figura 3.10.

3.7.3 Outros Aspectos

O desenvolvimento de software segundo uma abordagem orientada por objectos apresenta outras vantagens comparativamente com a abordagem estruturada, resumidamente:

- Providencia blocos de construção de alto nível que **reduzem os custos de desenvolvimento**, pela promoção da reutilização e encapsulamento de software. Este facto permite reduzir as interdependências e facilitar a manutenção posterior.
- Facilita a transferência de conhecimento através da adopção de “padrões de desenho”, **reduzindo o custo de aprendizagem**.
- Providencia um *framework* (aplicacional ou de *middleware*) para facilitar a extensão do sistema, **reduzindo o custo de novas versões**.
- **Reduz o custo de alteração** do sistema, em resposta às expectativas e requisitos dos utilizadores

Todavia, e ainda que actualmente esta abordagem possa já ser considerada estabelecida e madura, implica alguns cuidados na sua aplicação, pelo facto de levantar algumas questões:

- Exige uma nova forma de ”pensar“ o processo de desenvolvimento.
- A orientação por objectos deve ser utilizada em todas as fases do ciclo de vida do software. Por exemplo, não tem muito sentido usar-se uma linguagem orientada por objectos a seguir a um ciclo de análise e desenho tradicional, ou vice-versa.
- A gestão da empresa ou do projecto tem de “comprar” a mudança para a filosofia orientada por objectos, em particular tem de definir os objectivos comerciais para a mudança. Por exemplo, redução dos custos de manutenção do software.
- A migração para a orientação por objectos tem de ser devidamente planeada para garantir a sua eficácia.

Apesar das vantagens sublinhadas da abordagem da orientação por objectos no desenvolvimento de software, não se deve concluir que esta é “sempre” mais adequada que a aproximação estruturada. Há várias situações em que é mais correcto seguir uma metodologia estruturada. Por exemplo, se a linguagem de programação utilizada pela organização for Cobol, RPG ou C e o ambiente de desenvolvimento for estruturado, então fará mais sentido aplicar metodologias estruturadas ao

longo de todas as fases do projecto. Caso contrário, correr-se-ia no erro (e no perigo!) de se realizar uma modelação com base em conceitos orientados por objectos e depois ter de se recriar, ao nível do desenho (ou pior, ao nível da codificação!) um modelo completamente distinto do primeiro. Apesar de ser natural e óbvio que é mais consistente a utilização do mesmo paradigma ao longo de todo o ciclo, do ponto de vista teórico nada impede que se utilizem métodos de análise estruturada, seguidos de uma implementação numa linguagem orientada por objectos; o inverso pode também acontecer, isto é, a utilização de uma linguagem tradicional na sequência de uma análise orientada por objectos.

3.8 Conclusão

Este capítulo conclui a primeira parte do livro, na qual se deu uma visão dos problemas da Engenharia de Software (Capítulo 1), a definição de conceitos e abordagens genéricas ao desenvolvimento de software (Capítulo 2) e uma ideia resumida do caminho que foi seguido até à data em termos de evolução dos conceitos e das metodologias práticas (Capítulo 3).

Neste último capítulo procurámos transmitir ao leitor a ideia de que as abordagens orientadas por objectos apresentam vantagens significativas, mas que a proliferação de notações e metodologias, bem como o conhecimento especializado que é necessário para aplicar com sucesso esta nova forma de encarar e modelar o mundo exterior, colocam desafios significativos e constituem problemas importantes que é preciso solucionar.

É precisamente com esse objectivo que o resto do livro se concentra na linguagem de modelação UML, e nos processos de desenvolvimento a ela associados, quer porque esta é de facto uma iniciativa unificadora ao nível das notações, quer porque é necessária a sua divulgação de forma a que mais informáticos adquiram os conhecimentos necessários para a sua correcta utilização.

É preciso estar atento às iniciativas metodológicas mais recentes, nomeadamente àquelas que propõem uma simplificação do processo

de desenvolvimento ao nível de actividades, notações e documentação produzida. A grande maioria recorre ao UML para as actividades de modelação que realiza, o que mais uma vez reforça a nossa convicção que esta linguagem de notação representará cada vez mais um papel significativo nas metodologias de desenvolvimento de software.

3.9 Exercícios

Ex.16. Imagine que é o responsável de uma biblioteca cujos livros podem ser requisitados por diversos leitores previamente registados, mediante a apresentação de uma guia de requisição. Esta indica o prazo de devolução, ao fim do qual o leitor tem que devolver o livro, preenchendo uma guia de entrega. Para os livros muito solicitados poderão existir várias cópias na biblioteca, e se necessário, poderão ser adquiridas mais. Para tal, o responsável da biblioteca poderá requisitar mais cópias através de uma requisição, que será autorizada pela direcção, de modo a que o responsável da biblioteca possa encomendar esses livros junto de um fornecedor, proceder ao respectivo pagamento e dar entrada dos livros na biblioteca.

- Se tivesse que efectuar a análise deste caso segundo uma abordagem estruturada, que tipo de informação do enunciado seria mais relevante? E se a abordagem utilizada fosse orientada por objectos? Justifique a sua resposta.
- Efectue a modelação da estrutura estática do sistema pelo diagrama estruturado que achar mais conveniente, justificando a sua resposta.
- Efectue a modelação da estrutura estática do sistema pelo diagrama UML que achar mais conveniente, justificando a sua resposta.

Ex.17. Imagine que é o responsável pela gestão de informação sobre alunos de uma universidade, sendo para isso relevante reproduzir no sistema de informação as acções que normalmente ocorrem ao longo do ano na universidade: matrículas de alunos num ano lectivo e em cadeiras; inscrições nas aulas práticas; inscrições em exames; pagamentos de propinas; consultas de notas atribuídas; pedidos de certificados.

- Se tivesse que modelar o sistema anteriormente descrito, indique qual (ou quais) o(s) diagrama(s) que utilizaria se aplicasse uma abordagem estruturada e qual (ou quais) utilizaria se aplicasse uma abordagem orientada por objectos, justificando a sua resposta.
- Efectue a modelação dos conceitos aqui representados pelos diagramas estruturados que achar convenientes, justificando a sua resposta.
- Efectue a modelação dos conceitos aqui representados pelos diagramas relacionados com abordagens orientadas por objectos que achar convenientes, justificando a sua resposta. Quais as principais alterações que ocorreram no processo de desenvolvimento de software com a introdução das abordagens estruturadas?

Ex.18. Uma das críticas que normalmente se aponta às metodologias estruturadas é de defenderem conceitos teóricos que se afastam da realidade prática. Indique alguns exemplos deste desfasamento.

Ex.19. Compare os conceitos utilizados nos principais diagramas utilizados nas abordagens estruturadas e nas abordagens orientadas por objectos, para modelar a dinâmica de um sistema.

Ex.20. As abordagens orientadas por objectos, particularmente na tarefa de implementação, são particularmente adequadas para a implementação de processos iterativos. Justifique esta afirmação.

Ex.21. As metodologias estruturadas estão tipicamente divididas em duas categorias principais, consoante a importância que atribuem às principais técnicas de modelação utilizadas. Indique quais são essas categorias, e quais as principais características de cada uma deles.

Ex.22. Porque é que as metodologias de desenvolvimento de software estruturadas têm este nome? Quais as motivações que levaram à sua definição e aplicação?

Ex.23. Algumas das novas metodologias defendem o fim das actividades de análise no processo de desenvolvimento de software. Expresse a sua opinião, justificando-a.

Parte 2 – Linguagem de Modelação UML

- *Talvez isto seja um sinal – disse o Inglês, como quem pensa alto.*
- *Quem falou em sinais? – o interesse do rapaz crescia a cada minuto.*
- *Tudo na vida são sinais – disse o Inglês, desta vez fechando a revista que estava a ler. O universo é feito por uma língua que todo o mundo entende, mas que já se esqueceu. Estou à procura dessa Linguagem Universal, além de outras coisas. Por isso estou aqui.*

Paulo Coelho, *O Alquimista*.

O UML

O UML (*Unified Modelling Language*) é uma linguagem diagramática, utilizável para especificação, visualização e documentação de sistemas

de software. O UML surge em 1997 na sequência de um esforço de unificação de três das principais linguagens de modelação orientadas por objectos (OMT, Booch e OOSE). Seguidamente, adquiriu o estatuto de norma no âmbito da OMG e da ISO, tendo vindo a ser adoptado progressivamente pela indústria e academia em todo o mundo.

O UML apresenta, entre outras, as seguintes características principais: (1) é independente do domínio de aplicação (i.e., pode ser usado em projectos de diferentes características, tais como sistemas cliente/ servidor tradicionais; sistemas baseados na Web; sistemas de informação geográficos; sistemas de tempo real); (2) é independente do processo ou metodologia de desenvolvimento; (3) é independente das ferramentas de modelação; (4) apresenta mecanismos potentes de extensão; (5) agrega um conjunto muito significativo de diferentes diagramas/técnicas dispersos por diferentes linguagens (e.g., diagramas de casos de utilização, de classes, de objectos, de colaboração, de actividades, de estados, de componentes, e de instalação).

Ao longo dos Capítulos 4 a 9 da Parte 2 deste livro apresentamos a linguagem UML com um compromisso assumido entre abrangência e detalhe. Por um lado, pretende-se que sejam apresentados e discutidos todos os diagramas do UML; por outro lado, pretende-se apresentar os seus principais detalhes e discutir a sua aplicabilidade.

O UML e os Domínios de Aplicação

Os exemplos ilustrados e os exercícios sugeridos apresentam um compromisso entre situações de modelação de casos gerais do dia-a-dia (e.g., o exemplo da máquina de bebidas, ou os exercícios sobre o universo do futebol) e de casos mais específicos do domínio de sistemas de software (e.g., o exemplo dos diagramas de classes e de interacção do acesso, via JDBC, a um gestor de base de dados, ou o exercício pedido para modelar o ciclo de vida de um *servlet* Java). O objectivo de apresentar exemplos entre estes dois espectros é mostrar que o UML pode ser adoptado em diferentes domínios de aplicação, de abstracção e de generalidade.

O UML é uma linguagem gráfica cujo objectivo principal é promover e facilitar a comunicação entre um grupo variado de intervenientes.

Embora o foco deste livro tenha a ver com a modelação de software, nunca é demais sublinhar que o UML pode ser usado noutros contextos e por intervenientes com diferentes formações (e.g., por gestores, para representarem a organização das empresas e respectivos processos de negócio; por juristas, para representarem as relações entre leis).

Organização da Parte 2

A Parte 2 é constituída por 6 capítulos complementares. O Capítulo 4 dá a visão histórica e geral do UML e o Capítulo 9 descreve sucintamente alguns aspectos considerados “avançados”, não essenciais para o leitor que apenas pretende usar e aplicar as características básicas do UML. Os restantes capítulos (Capítulos 5, 6, 7 e 8) constituem o centro desta parte do livro e deverão ser lidos de forma sequencial, conforme proposto.

O Capítulo 4, “UML – Visão Geral”, dá uma visão geral do UML. Em particular apresenta o seu enquadramento histórico, os seus objectivos e principais aspectos que orientaram a sua arquitectura subjacente. São apresentados sucintamente os seus principais conceitos, designadamente os tipos de elementos básicos, de relações e de diagramas. Por fim, apresentam-se alguns dos mecanismos comuns UML (anotações e mecanismos de extensão) e considerações relacionadas com a organização dos artefactos de um sistema através da noção de pacotes.

O Capítulo 5, “UML – Casos de utilização”, começa por discutir a tarefa de especificação e de gestão de requisitos de um sistema de software e a sua relação com os diagramas de casos de utilização. São definidos os conceitos de caso de utilização, cenário, actor, relações entre casos de utilização (relação de generalização, inclusão ou extensão) e entre actores e casos de utilização (relação de comunicação). São apresentados modelos de especificação textual dos casos de utilização, com evidência para a especificação das relações de inclusão e de extensão. Por fim, é discutido a aplicação dos diagramas de utilização através do exemplo da “Máquina de Bebidas”.

O Capítulo 6, “UML – Modelação da Estrutura”, aborda um aspecto básico na modelação de software, que é a sua estrutura. Esta consiste,

segundo a abordagem orientada por objectos, na identificação de classes e suas respectivas relações. O UML providencia os seguintes elementos que permitem a especificação da estrutura de um sistema de software: classes, relações, interfaces, objectos. Com base nestes elementos, podem-se definir dois tipos de diagramas com fins complementares: diagramas de classes e diagramas de objectos.

O Capítulo 7, "UML – Modelação do Comportamento", parte do pressuposto que a modelação da estrutura de um sistema é conhecida, e atenta sobre a respectiva modelação de comportamento, a qual consiste, segundo a abordagem orientada por objectos, em dois tipos distintos de especificações. Por um lado, na modelação do comportamento entre objectos, pela identificação dos seus padrões de trocas de mensagens (diagramas de interacção). Por outro lado, na modelação do comportamento dentro de um objecto, pela identificação dos estados que um objecto pode ter ao longo do seu ciclo de vida e dos eventos que provocam mudanças nesses estados (diagramas de estados e de actividades).

O Capítulo 8, "UML – Modelação da Arquitectura", descreve aspectos da fase de implementação e instalação de um sistema de software, designadamente a estrutura e dependências de código fonte e de módulos executáveis, tal como a sua respectiva instalação nas diferentes plataformas computacionais subjacentes, através dos diagramas de componentes e de instalação. Os diagramas de componentes são usados para modelar a arquitectura de um sistema de software na perspectiva dos seus componentes digitais, explicitando principalmente as suas múltiplas dependências. Por outro lado, os diagramas de instalação são usados para modelar a arquitectura de um sistema informático da perspectiva dos seus componentes físicos explicitando as dependências de comunicação, e especificando os componentes instalados em cada nó computacional.

O Capítulo 9, "UML – Aspectos Avançados", apresenta alguns assuntos que consideramos "avançados", e que permitem ao utilizador do UML uma sua profunda compreensão e porventura uma sua melhor aplicação. Nomeadamente, apresentam-se os seguintes aspectos: (1) a estrutura e arquitectura do UML, e em particular a organização da camada de metamodelo; (2) os mecanismos de extensão do UML; com

alguns exemplos concretos de conjuntos de elementos que estendem o UML, designados por perfis; (3) considerações sobre as noções de componente, sistemas de componentes (*toolkits* e *frameworks*), famílias de aplicações, e reutilização; (4) tipos parametrizáveis em UML, em particular classes parametrizáveis e colaborações parametrizáveis (no contexto de padrões de desenho); e (5) uma breve introdução e motivação para o XMI, que é o standard OMG para interoperação de modelos UML, em XML.

Capítulo 4 - UML – VISÃO GERAL

Tópicos

- Introdução
- Visão Histórica
- Tipos de Elementos Básicos
- Tipos de Relações
- Tipos de Diagramas
- Mecanismos Comuns
- Tipos de Dados
- Organização dos Artefactos – Pacotes
- Exercícios

4.1 Introdução

O **UML** (***Unified Modeling Language***) é uma linguagem para especificação, construção, visualização e documentação de artefactos de um sistema de software. É promovido pelo Object Management Group (OMG), com contribuições e direitos de autoria das seguintes empresas: Hewlett-Packard, IBM, ICON Computing, i-Logix, IntelliCorp, Electronic Data Services, Microsoft, ObjecTime, Oracle, Platinum, Ptech, Rational, Reich, Softeam, Sterling, Taskon A/S e Unisys.

O UML providencia as seguintes particularidades principais [OMG99]:

- Semântica e notação para tratar um grande número de tópicos actuais de modelação.

- Semântica para tratar tópicos de modelação futura, relacionados em particular com a tecnologia de componentes, computação distribuída, *frameworks* e Internet.
- Mecanismos de extensão de modo a permitir que futuras aproximações e notações de modelação possam continuar a ser desenvolvidas sobre o UML.
- Semântica e sintaxe para facilitar a troca de modelos entre ferramentas distintas.

O UML alarga o âmbito de aplicações alvo comparativamente com outros métodos existentes, designadamente porque permite, por exemplo, a modelação de sistemas concorrentes, distribuídos, para a Web, sistemas de informação geográficos, etc.

A ênfase do UML é na definição de uma linguagem de modelação standard, e por conseguinte, o UML é independente das linguagens de programação, das ferramentas CASE, bem como dos processos de desenvolvimento. O objectivo do UML é que, dependendo do tipo de projecto, da ferramenta de suporte, ou da organização envolvida, devem ser adoptados diferentes processos/metodologias, mantendo-se contudo a utilização da mesma linguagem de modelação.

O UML é independente das ferramentas de modelação. Apesar da especificação do UML incluir sugestões para os fabricantes de ferramentas adoptarem na apresentação dessas notações (e.g., tópicos como o desenho de diagramas, cor, navegação entre esquemas, etc.), não aborda todos os requisitos necessários por não ser esse, propositadamente, o seu objectivo.

Um princípio básico no esforço de definição do UML foi a simplicidade. Outro princípio foi a coerência na unificação de diferentes elementos existentes em vários métodos, entre outros Booch [Booch94], OMT [Rumbaugh91] e OOSE [Jacobson92], e introdução de novos elementos apenas quando tal fosse absolutamente necessário, i.e., quando tais elementos não existissem em métodos anteriores.

Os novos elementos introduzidos no UML são:

- Mecanismos de extensão
- Elementos para modelar processos e *threads*
- Elementos para modelar distribuição e concorrência

- Padrões de desenho e colaborações
- Diagramas de actividades (para modelação de processos de negócio)
- Refinamento (para tratar relações entre diferentes níveis de abstracção)
- Interfaces e componentes
- Linguagem de restrições (*Object Contraint Language*)

O UML providencia os seguintes tipos de diagramas:

- Diagramas de casos de utilização
- Diagramas de classes e diagramas de objectos
- Diagramas de comportamento
  - Diagramas de estados (*statechart*)
  - Diagramas de actividades
  - Diagramas de interacção (diagramas de sequência e diagramas de colaboração)
- Diagramas de arquitectura:
  - Diagramas de componentes
  - Diagramas de instalação

4.2 Visão Histórica

A Figura 4.1 dá uma visão do enquadramento histórico relativamente ao contexto das linguagens de modelação de software orientado por objectos e, em particular, do UML.

Na primeira metade da década de 1990 assistiu-se a uma proliferação de métodos e notações para modelação segundo a abordagem orientado por objectos (fase da fragmentação). Por essa altura percebeu-se a necessidade da existência de uma linguagem que viesse a tornar-se uma norma, que fosse aceite e utilizada quer pela indústria, quer pelos ambientes académicos e de investigação.

Surgiram entretanto alguns esforços nesse sentido de normalização, sendo que o UML apareceu em 1996 melhor posicionado como a linguagem "unificadora" de notações, diagramas, e formas de representação

existentes em diferentes métodos, em particular nos métodos Booch, OMT e OOSE (fase da unificação).

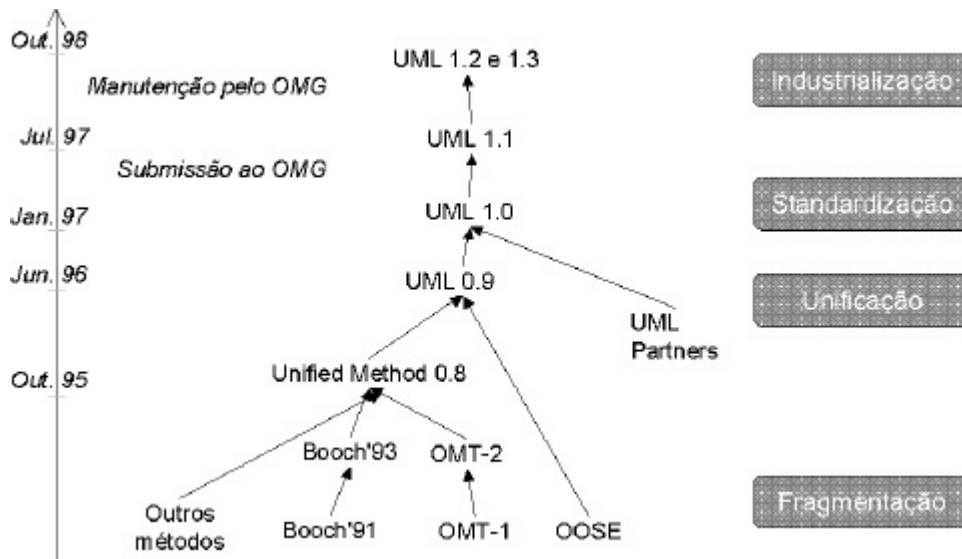



Figura 4.1: Visão histórica do UML.

Seguiu-se um esforço significativo nessa unificação com contributos de vários parceiros com vista à normalização no âmbito da OMG, o que aconteceu em 1997 (fase da standardização).

Actualmente assiste-se à divulgação e adopção generalizada do UML como "a" linguagem de modelação de software segundo a abordagem orientada por objectos. Assiste-se ao aparecimento de publicações específicas sobre UML; de teses, relatórios e artigos técnico-científicos que usam o UML; de ferramentas CASE que suportam o UML, etc. (fase da industrialização).

Há dois aspectos importantes que se obtêm com o UML: (1) terminam as diferenças, geralmente inconsequentes, entre as linguagens de modelação dos anteriores métodos; e (2) unifica as distintas perspectivas entre diferentes tipos de sistemas (e.g., modelação de negócio vs. modelação de software), fases de um processo e conceitos internos.

Uma das preocupações mais significativos no desenho e na especificação do UML foi torná-lo extensível e aberto a futuras evoluções, que

inevitavelmente irão ocorrer. O objectivo é que seja possível estender o UML sem ser necessário redefinir o seu núcleo principal.

4.3 Tipos de Elementos Básicos

A estrutura de conceitos do UML é razoavelmente abrangente consistindo num conjunto variado de notações, as quais podem ser aplicados em diferentes domínios de problemas e a diferentes níveis de abstracção. A estrutura de conceitos do UML pode ser vista através das seguintes noções: (1) "coisas" ou elementos básicos, com base nos quais se definem os modelos; (2) relações, que relacionam elementos; e (3) diagramas, que agrupam elementos.

Os elementos encontram-se organizados consoante a sua funcionalidade ou responsabilidade. Assim há elementos de estrutura, comportamento, agrupamento e de anotação.

A Figura 4.2. ilustra o conjunto dos principais elementos de estrutura: classes, classes activas, interfaces, casos de utilização, actores, colaborações, componentes e nós.

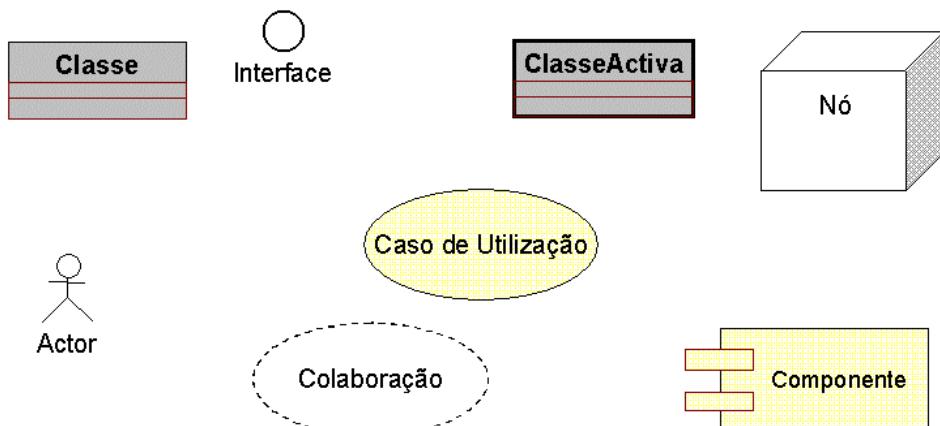



Figura 4.2: Resumo dos elementos de estrutura.

A Figura 4.3 ilustra outros elementos básicos do UML, elementos de comportamento (estados e mensagens), de agrupamento (pacotes) e de anotação (anotações ou notas).

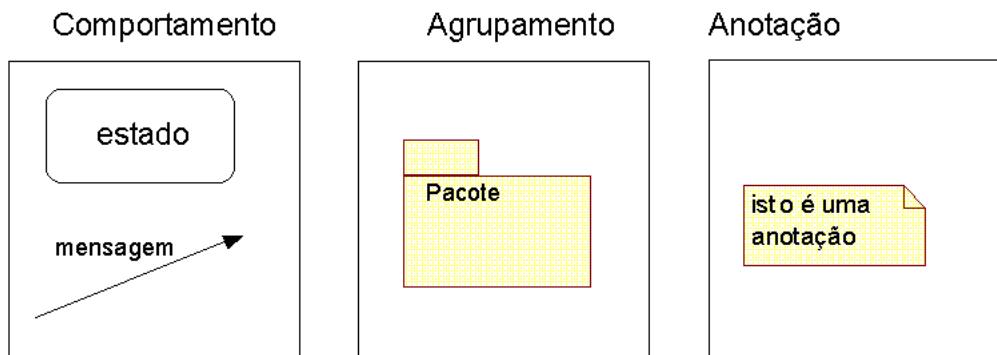



Figura 4.3: Resumo dos elementos de comportamento, agrupamento e anotação.

4.4 Tipos de Relações

As relações são conceitos gerais que apresentam uma sintaxe (neste caso, uma notação) e uma semântica bem definida, e que permite o estabelecimento de interdependências entre os elementos básicos acima introduzidos.

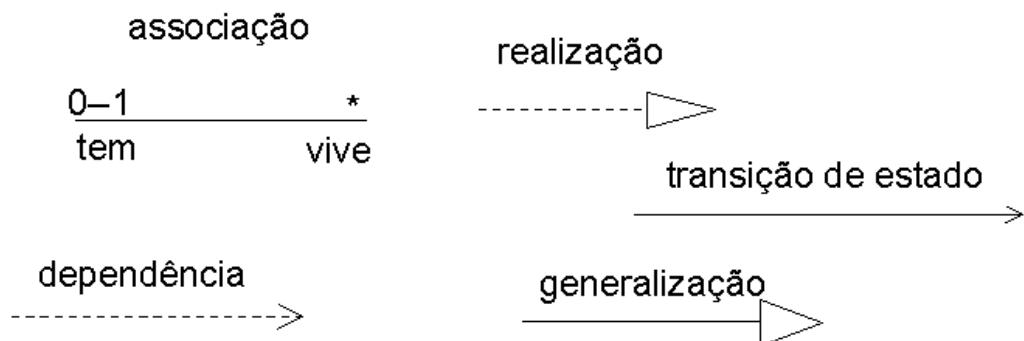



Figura 4.4: Resumo dos tipos de relações standard.

A Figura 4.4 ilustra os principais tipos de relações do UML, nomeadamente relações do tipo associação, dependência, realização,

generalização e transição de estado. (Mais abaixo será descrito em detalhe a semântica e aplicação destes diferentes tipos de relações.)

4.5 Tipos de Diagramas

Os diagramas são conceitos que traduzem a possibilidade de agrupar elementos básicos e suas relações de uma forma lógica ou de uma forma estrutural. Existem diferentes tipos de diagramas em UML. Em cada tipo de diagrama é usado um subconjunto dos elementos básicos acima descritos, com diferentes tipos de relações que tenha sentido existir.

Por exemplo, um diagrama de classes UML é composto por um conjunto de ícones representantes de classes em simultâneo (e opcionalmente) com a representação explícita das suas relações.

Com base no quarto princípio de modelação enumerado na Secção 2.3.2 ("P4. Nenhum modelo é suficiente por si só. Qualquer sistema não-trivial é melhor representado através de pequeno número de modelos razoavelmente independentes".), o UML define diferentes tipos de diagramas, cuja utilização e aplicação permitem dar visões complementares.

- Diagramas de casos de utilização, que representam a visão do sistema na perspectiva do seu utilizador.
- Diagramas de classes que permitem especificar a estrutura estática de um sistema segundo a abordagem orientada por objectos.
- Diagramas de interacção entre objectos (diagramas de sequência e diagramas de colaboração) e diagramas de transição de estados e diagramas de actividades, que permitem especificar a dinâmica ou o comportamento de um sistema segundo a abordagem orientada por objectos.
- Diagramas de componentes e diagramas de instalação, que dão uma visão da disposição dos componentes físicos (software e hardware) de um sistema.

4.5.1 Diagramas de Casos de Utilização

Um diagrama de casos de utilização descreve a relação entre actores e casos de utilização de um dado sistema (ver exemplo da Figura 4.5). Este é um diagrama que permite dar uma visão global e de alto nível do sistema, sendo fundamental a definição correcta da sua fronteira.

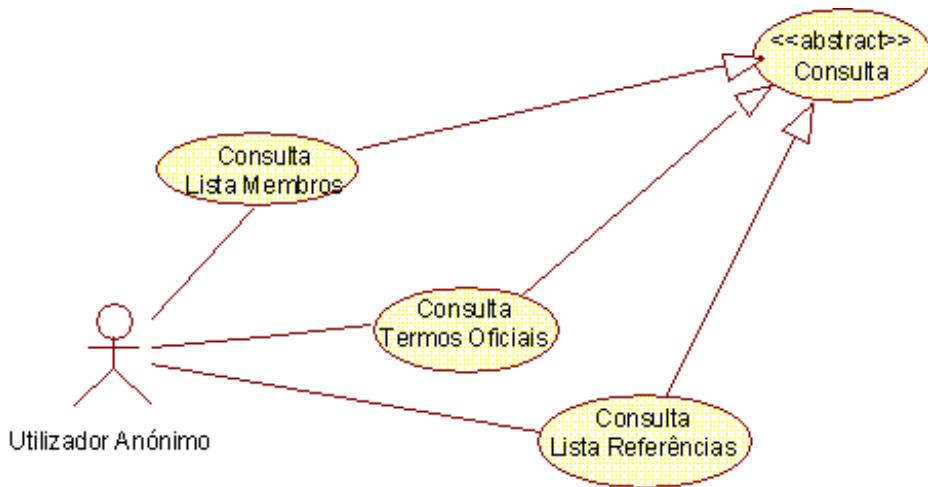



Figura 4.5: Exemplo de um diagrama de casos de utilização.

Estes diagramas são utilizados preferencialmente na fase de especificação de requisitos e na modelação de processos de negócio. Estes diagramas são equivalentes aos homólogos existentes no método OOSE de Ivar Jacobson [Jacobson92].

4.5.2 Diagramas de Modelação da Estrutura

Os diagramas de classes do UML são uma integração de diferentes diagramas de classes existentes, nomeadamente no OMT [Rumbaugh91], Booch [Booch94] e outros métodos OO. Extensões específicas de determinados processos (e.g. recorrendo a estereótipos e correspondentes ícones) podem ser definidos em vários diagramas para suportarem diferentes estilos de modelação.

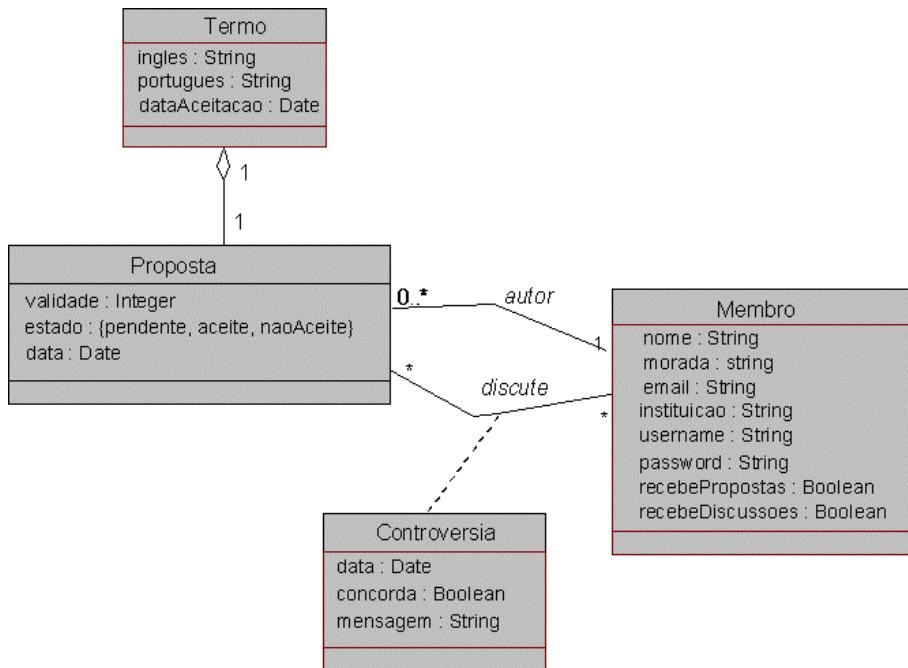



Figura 4.6: Exemplo de um diagrama de classes.

Os diagramas de classes (ver exemplo da Figura 4.6) descrevem a estrutura estática de um sistema, em particular as entidades existentes, as suas estruturas internas, e relações entre si.

Um diagrama de objectos descreve um conjunto de instâncias compatíveis com determinado diagrama de classes. Permitem ilustrar os detalhes de um sistema em determinado momento ao providenciarem cenários de possíveis configurações.

4.5.3 Diagramas de Modelação do Comportamento

Interacção entre Objectos

Os padrões de interacção entre objectos são representados por diagramas de interacção, os quais podem assumir duas formas complementares, cada um focando um aspecto distinto: diagramas de sequência e diagramas de colaboração.

Diagramas de Sequência

Os diagramas de sequência ilustram interacções entre objectos num determinado período de tempo (ver exemplo da Figura 4.7). Em particular, os objectos são representados pelas suas "linhas de vida" e interagem por troca de mensagens ao longo de um determinado período de tempo.

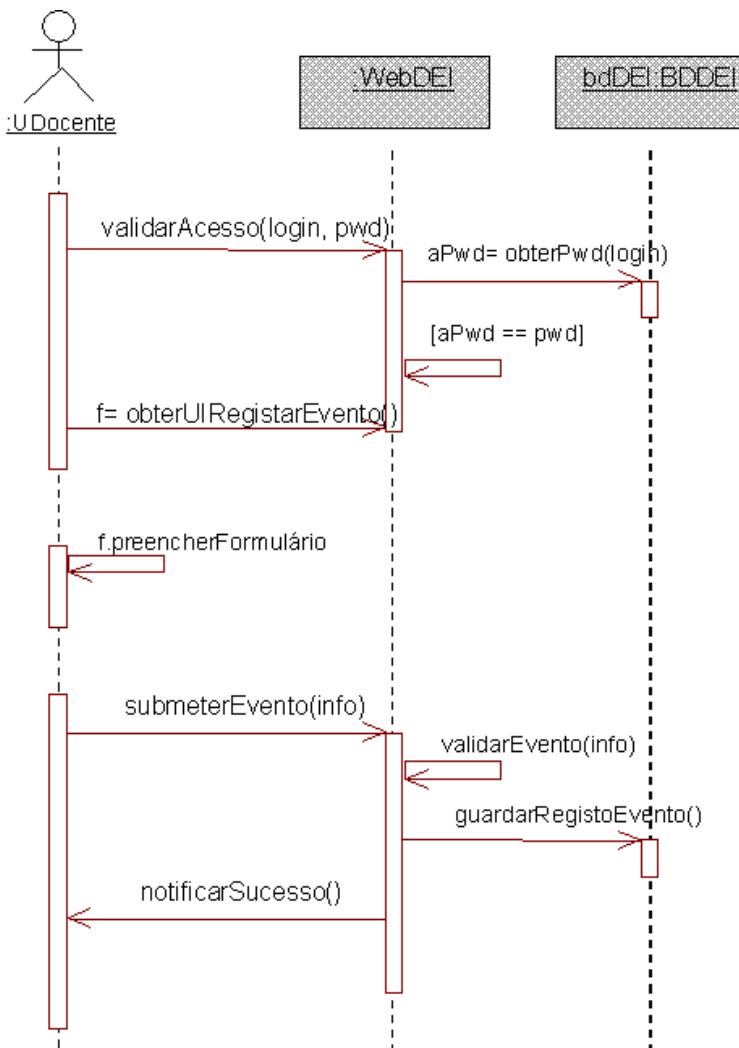



Figura 4.7: Exemplo de um diagrama de sequência.

Este tipo de diagrama baseia-se numa variedade de diagramas homólogos existentes em distintos métodos OO, segundo diferentes designações, como por exemplo *interaction diagrams*, *message sequence charts*, *message trace, event trace*, etc.

Diagramas de Colaboração

Os diagramas de colaboração ilustram interacções entre objectos com ênfase para a representação das ligações entre objectos. Como os diagramas de colaboração não mostram o tempo como um elemento explícito (tal como acontece nos diagramas de sequência), a sequência de mensagens e de actividades concorrentes é determinada usando-se números sequenciais, com diferentes níveis hierárquicos. Colaborações são entidades de modelação principais e constituem geralmente a base para a representação visual de padrões de desenho (*design patterns*) [Gamma94].

Este tipo de diagrama foi adoptado, entre outros, a partir do diagrama de objectos de Booch [Booch94], e do diagrama de interacção de objectos de Fusion [Malan96].

Diagramas de Transição de Estado

Os diagramas de transição de estado descrevem as sequências de estados que um objecto ou uma interacção pode passar ao longo da sua existência em resposta a estímulos recebidos, conjuntamente com as suas respostas e acções.

São baseados nos *statecharts* de David Harel com pequenas adaptações [Harel87, Harel96].

Diagramas de Actividades

Os diagramas de actividades (ver exemplo da Figura 4.8) são um caso particular dos diagramas de transição de estado, no qual a maioria dos estados são substituídos pelos conceitos correspondentes a acções e/ou actividades, e no qual as transições são desencadeadas devido à conclusão de acções nos estados originais.

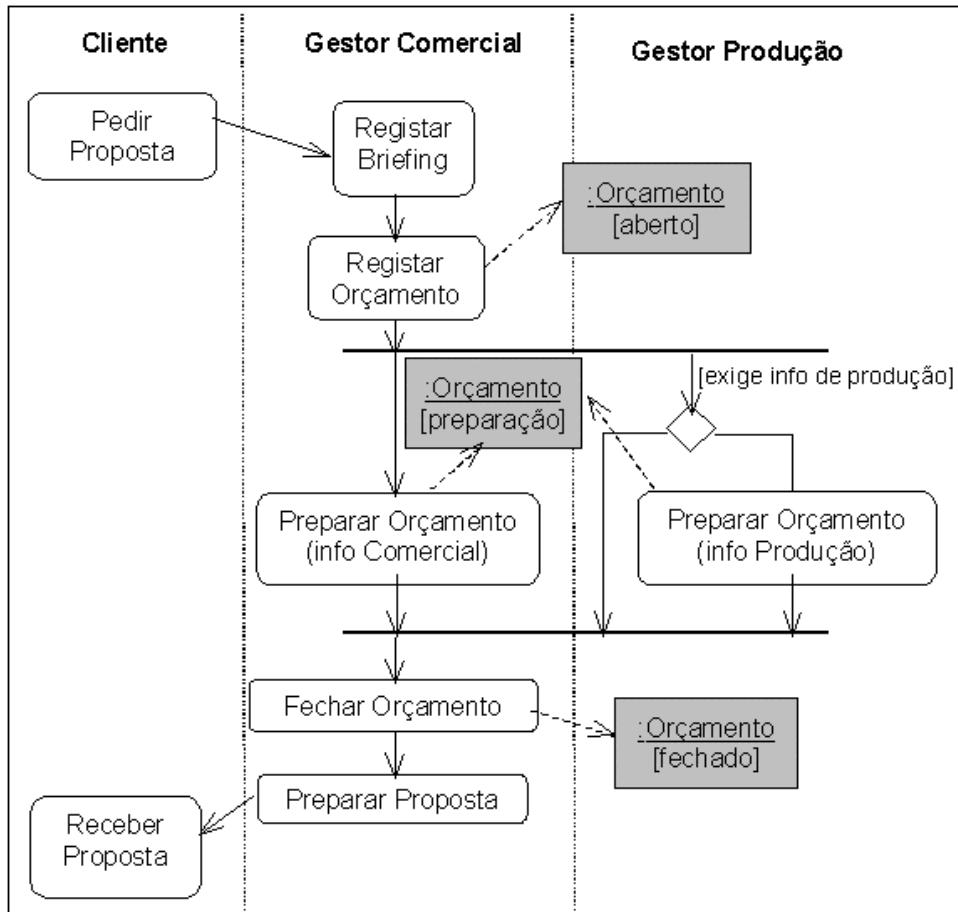
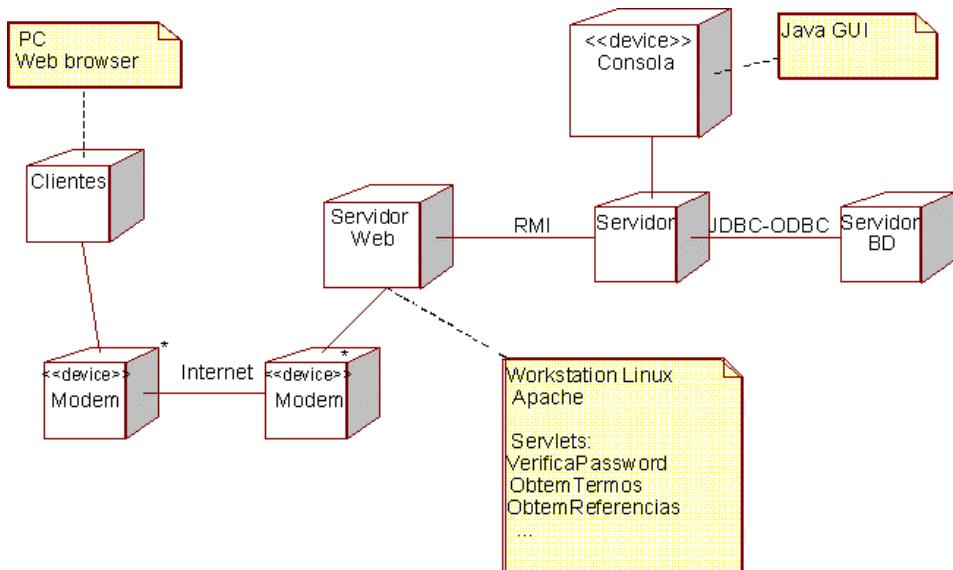



Figura 4.8: Exemplo de um diagrama de actividades.

O objectivo deste diagrama é representar os fluxos conduzidos por processamento interno, em contraste com eventos externos representados tipicamente nos diagramas de estado.

Este tipo de diagramas pode ser encarados como um caso particular de diagramas de estados e inspirados nos diagramas de *workflow*, fluxogramas ou naqueles baseados em SDL (*Specification and Description Language*) [ITU-T93].

4.5.4 Diagramas de Arquitectura

Os diagramas de arquitectura descrevem aspectos da fase de implementação de um sistema de software, por exemplo, relativamente à estrutura e dependências de módulos de código fonte e de módulos executáveis. Estes diagramas apresentam-se sob duas formas: diagramas de componentes e diagramas de instalação.

Diagramas de componentes

Os diagramas de componentes descrevem as dependências entre componentes de software, incluindo componentes de código fonte, código binário e executáveis.

Os diagramas de componentes são representados na forma de tipos e não na forma de instâncias. Para descrever-se instâncias de componentes, usam-se os diagramas de instalação.

Diagramas de Instalação

Os diagramas de instalação (*deployment diagrams*) descrevem a configuração de elementos de suporte de processamento, e de componentes de software, processos e objectos existentes nesses elementos (ver exemplo da Figura 4.9).

Figura 4.9: Exemplo de um diagrama de instalação.

4.6 Mecanismos Comuns

O UML contém um conjunto de mecanismos comuns que são aplicados de modo consistente ao longo dos seus diferentes diagramas. Descreve-se de seguida dois dos mecanismos comuns mais usados: anotações e mecanismos de extensão.

4.6.1 Notas (Anotações)

Notas ou anotações são os adornos mais importantes que existem autonomamente (i.e., sem estarem directamente associados a outros elementos). Ver-se-á mais à frente a utilização de outros adornos (por exemplo, adornos para qualificar elementos participantes em relações).

Uma **nota** ilustra um comentário e não tem qualquer impacto semântico, já que o seu conteúdo não altera o significado do modelo no qual ela se encontra (ver Figuras 4.9 e 4.10). Por isso é normal a utilização de notas para se descrever informalmente: requisitos, restrições, observa-ções, revisões ou explicações.

Considerar a aplicação do padrão de desenho `Singleton` e `Factory`.

alb. 2000/6/12

Ver `http://www.objectspace.com` para mais informação sobre o ORB Voyager

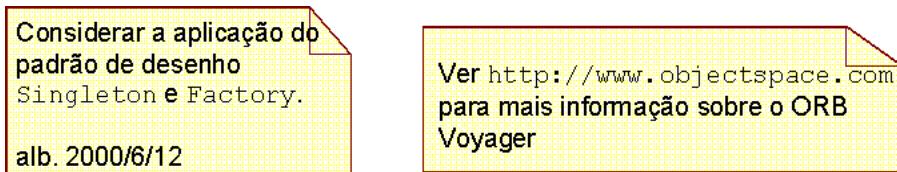



Figura 4.10: Exemplo de notas.

Em geral devem ser tomadas em consideração as seguintes observações na utilização de notas:

- Localização: Devem ser colocadas graficamente perto dos elementos que descrevem.
- Visibilidade: Podem-se esconder ou tornar visíveis (isto dependendo das ferramentas de suporte).
- Extensão: Devem-se usar referências (e.g., nomes de documentos ou URL) caso se pretenda um comentário mais extenso.
- Evolução: Há medida que o modelo evolui as notas mais antigas (cuja relevância e utilidade deixem de ter sentido) devem ser eliminadas dos modelos.

4.6.2 Mecanismos de Extensão

O UML providencia os seguintes mecanismos que permitem estender a sua linguagem de forma consistente: estereótipos, marcas e restrições. Apresentam-se na Secção 9.3 mais detalhes sobre este assunto. Interessa desde já referir alguns aspectos na aplicação destes mecanismos, designadamente deve-se:

- Introduzir um número reduzido destes elementos.
- Escolher nomes curtos e com significado para estereótipos e marcas.
- Sempre que a precisão puder ser relaxada, usar texto livre para especificação das restrições. Caso contrário, usar a linguagem OCL. O OCL (*Object Constraint Language*) [Warmer99] é uma linguagem para especificação formal de restrições e é parte integrante do UML [OMG99].

- A definição destes mecanismos de extensão do UML, em particular a introdução de estereótipos, deve ser realizada por um número restrito de especialistas em UML e deve ser devidamente documentado.

Estereótipos

Um **estereótipo** é um metatipo, isto é, um tipo que descreve tipos. Permite definir novos tipos de elementos sem se alterar o metamodelodo do UML, e por conseguinte permite estender o UML. O nome de um estereótipo é representado entre os caracteres '«' e '»'. Exemplos:

- Na modelação de processos de negócio: «trabalhador», «documento», «política».
- Na modelação de aplicações específicas: classes de «interface», «controlo», e «entidade».
- Na modelação de aplicações Web, usando o *Web Application Extension* [Conallen00]): classes de «Server Page», «Client Page», «Form», «Select Element».

Figura 4.11: Exemplo de estereótipos.

A Figura 4.11 ilustra três formas complementares de apresentação de estereótipos: com ícone (e.g., `SensorTemperatura`); com nome (e.g., `ModelElement`); e com nome e ícone (e.g., `Overflow`).

A definição de um estereótipo tem de ter em conta os seguintes aspectos:

- Propriedades (pode providenciar o seu próprio conjunto de marcas)
- Semântica (pode providenciar as suas próprias restrições)
- Notação (pode providenciar o seu próprio ícone)

- A classe base do metamodelo que vai ser estendida (i.e., se o estereótipo estende uma classe, uma associação, um componente, etc.)

Marcas com Valor

Cada elemento em UML tem um conjunto de propriedades. Por exemplo: as classes têm um nome, uma lista de atributos e uma lista de operações; as associações têm um nome e dois ou mais elementos participantes. Enquanto que os estereótipos permitem adicionar novos elementos ao UML, as **marcas com valor** permitem adicionar novas propriedades aos elementos, quer sejam elementos já existentes no UML, quer sejam elementos definidos por recurso a novos estereótipos.

Uma marca com valor é um conceito que deve ser entendido como metadata (isto é, dados que descrevem dados) pois o seu valor aplica-se ao próprio elemento e não às suas instâncias. Por exemplo, conforme ilustrado na Figura 4.12, pode-se especificar o número de processadores instalados em cada tipo de nó, ou pode-se especificar se um determinado componente é para ser instalado/usado com perfil de cliente, servidor, ou ambos.

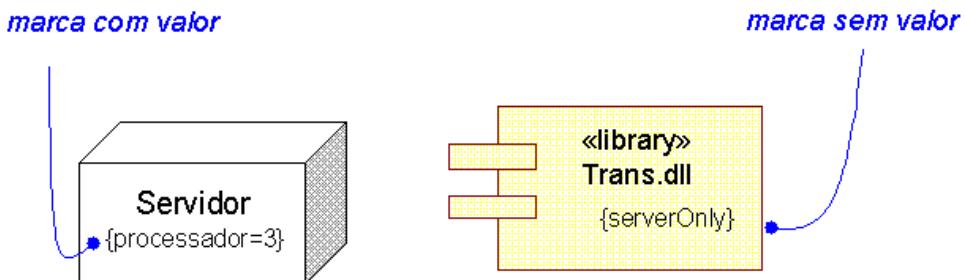



Figura 4.12: Exemplo de marcas com valor.

Um par “marca” e “valor” é delimitado entre os caracteres ‘{’ e ‘}’. Exemplos de aplicações usuais:

- Para geração de código: E.g.: {language=Java}, {linker=Blinker}.
- Na produção automática de documentação.
- Na gestão de configurações: E.g.: {autor=AMRS}, {data=...}.

Restrições

As **restrições** (*constraints*) permitem adicionar ou alterar a semântica assumida por omissão de qualquer elemento UML. Uma restrição consiste na especificação de uma condição delimitada pelos caracteres '{' e '}'. A condição pode ser especificada numa linguagem formal (e.g., OCL) ou informal (e.g., texto em português).

Uma restrição permite especificar condições que têm de ser validadas para que o modelo esteja "bem definido".

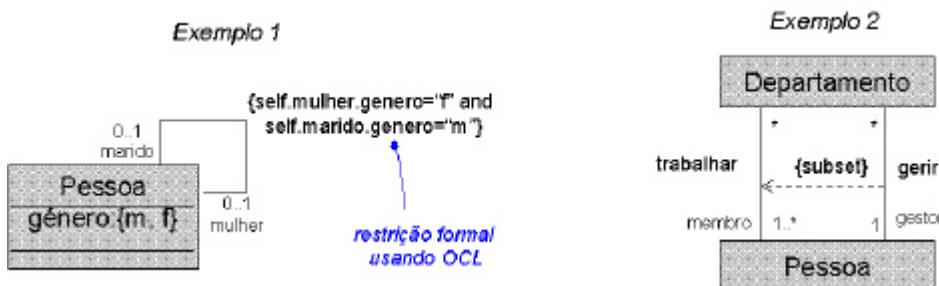



Figura 4.13: Exemplos de utilização de restrições.

A Figura 4.13 ilustra dois exemplos de especificação de restrições. No primeiro exemplo especifica-se em OCL que "*uma pessoa pode estar casada apenas com outra pessoa do sexo oposto*" (Exemplo 1). No segundo exemplo, especifica-se através da restrição `{subset}`, predefinida em UML, que os elementos da associação "gerir" tem de existir obrigatoriamente na associação "trabalhar", ou seja especifica-se que "*uma pessoa para ser gestor de um departamento tem também de ser, necessariamente, membro desse departamento*" (Exemplo 2).

4.7 Tipos de Dados

Um **tipo de dado** é uma abstracção utilizada de forma implícita no UML. Os tipos de dados não são elementos do modelo e por conseguinte não lhe são associados estereótipos, marcas com valor ou restrições. Um tipo primitivo é um tipo de dados que não tem uma subestrutura.

Exemplos de tipos de dados:

- Primitivos: `Integer`, `String`, `Time`

- Enumerados: Boolean, AggregationKind, VisibilityKind
- Outros: Expression, Mapping, Name, Multiplicity

Note-se que os conceitos de nomes, expressões ou multiplicidade são tipos de dados UML, definidos informalmente à custa de outros tipos primitivos. Por exemplo, o tipo `Multiplicity` é definido como "o conjunto não vazio de inteiros (`Integer`) positivos, estendido com o caracter '*'"; o tipo `Expression` é definido como "uma sequência de caracteres (`String`) cuja sintaxe não é definida, propositadamente, pelo UML".

Para mais detalhes consulte-se a Secção 9.2.2.

4.8 Organização dos Artefactos – Pacotes

Um **pacote** (*package*) é em UML um elemento meramente organizacional. Permite agregar diferentes elementos de um sistema em grupos de forma que semântica ou estruturalmente faça sentido.

Um pacote pode conter outros elementos, incluindo: classes, interfaces, componentes, nós, casos de utilização, e mesmo outros pacotes. Por outro lado, um elemento encontra-se definido em apenas um único pacote.

A necessidade da existência de pacotes torna-se evidente na modelação de sistemas de média/grande dimensão, em que, por razões de ordem prática, se torna impossível modelizá-los de uma "só vez". Os principais benefícios da sua utilização são: (1) facilita a gestão e procura de artefactos; (2) evita os conflitos de nomes (e.g., X::A é diferente de X::Y:A, e diferente de Z::A); e (3) providencia um mecanismo de controlo de acessos (visibilidade).

Elementos de diferentes tipos podem ter o mesmo nome dentro de um pacote. Por exemplo, pode existir num pacote uma classe e um componente com o nome `Entidade`. Contudo, de modo a reduzir ou eliminar as possíveis confusões, sugere-se que tal prática não seja adoptada.

4.8.1 Representação Gráfica

Em UML os pacotes são representados graficamente por uma pasta (*tabbed folder*) com duas zonas complementares: um pequeno rectângulo (designado por tabulador ou *tab*), normalmente com o nome do pacote; e um rectângulo maior onde normalmente se especificam os elementos constituintes desse pacote ou, pelo menos, os seus elementos públicos.

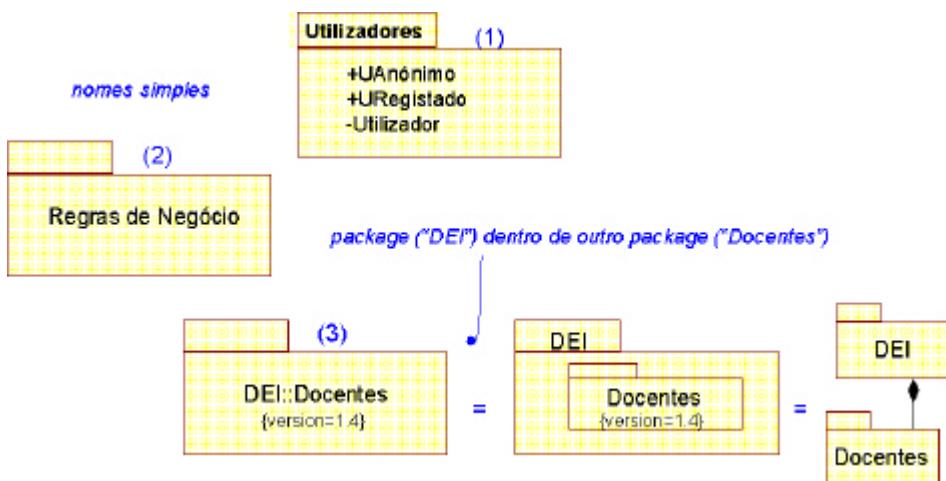



Figura 4.14: Exemplos de pacotes.

A Figura 4.14 ilustra alguns exemplos de representação de pacotes. No exemplo (1) o pacote tem o nome `Utilizadores` e descreve os seus elementos. Os símbolos "+", "-" e "#" representam informação de visibilidade, respectivamente visibilidade pública, privada e protegida (ver secção seguinte para mais detalhes).

O exemplo (2) ilustra uma forma alternativa de representar um pacote em que não são apresentados os seus elementos. O nome do pacote é `Regras de Negócio` e encontra-se colocado no meio do rectângulo maior.

Por fim, o exemplo (3) ilustra dois aspectos interessantes. Por um lado, a possibilidade de relações de hierarquia ou de inclusão entre pacotes: o pacote `Docentes` está contido no pacote `DEI` e pode ser identificado univocamente pela concatenação dos vários nomes separados pelo

símbolo "::". Outro aspecto interessante é a possibilidade de caracterizar o pacote através dos mecanismos comuns discutidos na Secção 4.6, tais como especificação de marcas (e.g., `{version=1.4}`), estereótipos ou restrições.

4.8.2 Relações entre Pacotes

Os pacotes apresentam entre si diferentes tipos de relações, em particular relações de importação, exportação e generalização.

Visibilidade

Os símbolos "+", "-" e "#" (ou similares, que as ferramentas CASE proponham), à semelhança do que acontece na especificação de atributos e operações de classes, permitem indicar o tipo de **visibilidade** que os elementos constituintes de um pacote apresentam.

- Um elemento com **visibilidade pública** (prefixado com o símbolo "+") pode ser usado/referenciado por qualquer outro elemento independentemente do local onde é definido.
- Um elemento com **visibilidade privada** (prefixado com o símbolo "-") pode ser usado/referenciado por elementos definidos no mesmo pacote.
- Um elemento com **visibilidade protegida** (prefixado com o símbolo "#") pode ser usado/referenciado por um elemento definido no mesmo pacote ou num outro pacote que seja uma especialização (através da relação de herança) do primeiro.

À semelhança do que acontece com algumas linguagens de programação (e.g., C++) relativamente à relação entre classes, é possível definir-se uma relação de *friend* entre dois pacotes (é uma relação de dependência entre pacotes, com estereótipo «friend»). Neste caso, um pacote que é *friend* de outro pode aceder/referenciar todos os seus elementos independentemente da sua visibilidade. Contudo este tipo de dependência deve ser evitado sempre que possível porque viola os princípios do encapsulamento e da minimização de dependências.

Importação e Exportação

Um pacote faz a **exportação**, por definição, de todos os seus elementos públicos. Mas tal facto não implica que um elemento definido noutro pacote possa aceder/referenciar um elemento exportado num outro pacote. Para que tal pudesse ocorrer deveria existir explicitamente uma relação de importação entre esses dois pacotes.

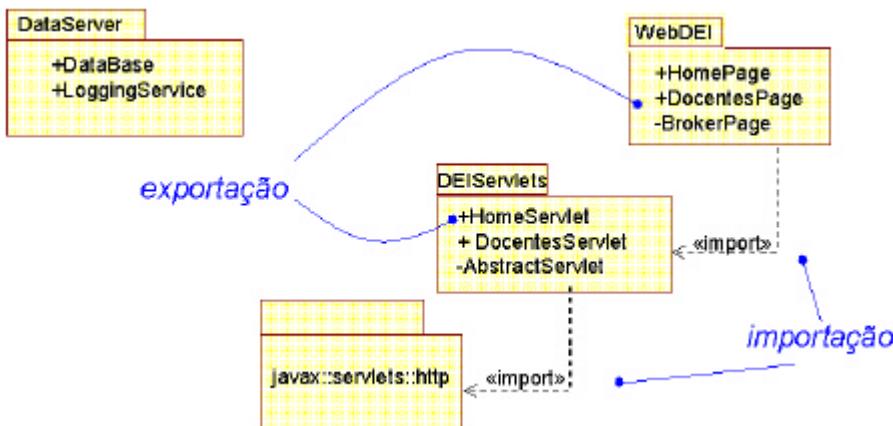



Figura 4.15: Relações de importação/exportação entre pacotes.

Por exemplo, o elemento `DataBase` exportado no pacote `DataServer` não pode ser referenciado por qualquer elemento definido em `WebDEI` (ver Figura 4.15).

A relação de **importação** é uma relação de dependência entre pacotes, especificando que o pacote base importa todos os elementos públicos definidos no pacote destino, ou seja, que os elementos públicos definidos no pacote destino podem ser usados por elementos no pacote base. A relação de importação é representada graficamente através de uma linha dirigida, a tracejado, com estereótipo «import».

Por exemplo, segundo a Figura 4.15, o pacote `WebDEI` importa (todos os elementos públicos definidos em) `DEIServlets`, e este por sua vez importa todos os elementos públicos definidos em `javax::serlets::http`.

Note-se que a relação de importação não é transitiva. No exemplo da Figura 4.14, o `WebDEI` importa `DEIServlets`, e este importa

`javax::serlets::http`, no entanto `WebDEI` não importa o `javax::serlets::http`. Isto significa que os elementos definidos em `WebDEI` podem aceder aos elementos públicos de `DEIServlets`, mas não aos de `javax::serlets::http`. Para que tal fosse possível, dever-se-ia definir explicitamente uma relação de importação entre esses dois pacotes.

Note-se, por fim, que a relação de importação não é simétrica. Ou seja, o facto dos elementos definidos em `WebDEI` poderem aceder aos elementos públicos de `DEIServlets` não implica que o inverso seja verdade.

Preferencialmente, os elementos exportados por cada pacote devem ser do tipo interface, uma vez que providenciam uma interface de programação sem revelarem os detalhes de implementação e as relações de classes definidas internamente no respectivo pacote.

Generalização

A relação de generalização entre pacotes é semelhante à homóloga existente entre classes, e é usada para a especificação de famílias de pacotes, típicas em sistemas complexos ou flexíveis (e.g., significativamente parametrizáveis, multi-plataforma, multi-linguagem).

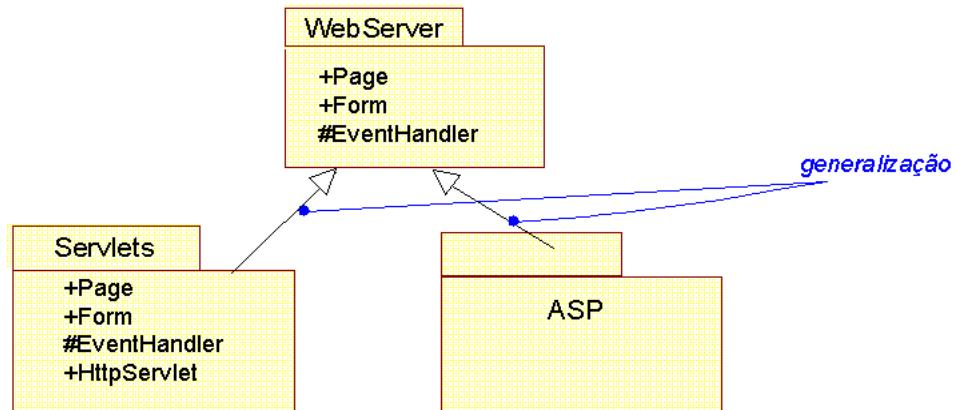



Figura 4.16: Relações de generalização entre pacotes.

A Figura 4.16 ilustra uma relação de generalização entre pacotes. O pacote `WebServer` tem duas classes públicas (`Page` e `Form`) e uma protegida (`EventHandler`) e é especializado por dois pacotes distintos: `Servlets`, que especializa o pacote de topo segundo a tecnologia da Sun (Java Servlets); e ASP, que providencia a mesma funcionalidade segundo a tecnologia da Microsoft (Active Server Pages).

4.8.3 Tipos de Pacotes

Na generalidade dos exercícios académicos a dimensão e/ou a simplicidade do problema faz com que um pacote seja "simplesmente" um pacote. Contudo, em situações reais de modelação de sistemas de software de média/grande dimensão, realizada por equipas de vários indivíduos, torna-se necessário tipificar os próprios pacotes.

A especificação actual do UML propõe cinco estereótipos standard aplicados a pacotes:

- «system»: pacote que representa o sistema inteiro; tipicamente este pacote agrega pacotes dos restantes tipos (subsistema, fachada, etc.).
- «subsystem»: pacote que representa uma parte constituinte do sistema inteiro.
- «facade»: pacote com elementos (tipicamente classes e interfaces) que constituem a fachada (ou a interface de programação) providenciada conjunta e coerentemente por outros pacotes. A fachada permite esconder os detalhes de arquitectura e de implementação de vários elementos eventualmente organizados em distintos pacotes. Os elementos definidos numa fachada apenas referem outros elementos definidos noutros pacotes.
- «framework»: um *framework* é uma arquitectura de classes e interfaces com inúmeros pontos de variabilidade ou de extensão e com estruturas de objectos padronizadas, conhecidas por padrões de desenho. O desenvolvimento de sistemas baseados em *frameworks* e em componentes de software é um tópico emergente extremamente actual.

- «stub»: um adaptador (*stub)* é usado quando se pretende partir um sistema em diferentes pacotes por motivos de divisão de trabalho, quer em termos físicos/espaciais, quer em termos temporais. Os adaptadores permitem que duas equipas consigam trabalhar paralelamente em diferentes locais, mas mantendo algum nível de interdependência.

Em geral os pacotes mais usados são do tipo sistema e subsistema, podendo contudo, surgir situações em que um pacote é claramente do tipo fachada, adaptador ou *framework*. Por omissão assume-se que um pacote é do tipo sistema (se for de primeiro nível) e subsistema (se for de segundo ou mais nível). Recomenda-se a consulta da Secção 9.5 para uma melhor compreensão dos tipos de pacotes acima referidos, em particular, dos conceitos de *framework*, fachada e famílias de aplicação.

4.8.4 Modelação de Grupos de Elementos

Compete a cada processo de desenvolvimento de software formalizar ou dar sugestões relativamente à forma de organizar todo o processo através de uma estrutura adequada de pacotes. Este aspecto é analisado na Parte 3 do livro.

Referem-se, no entanto, algumas das formas clássicas de organização dos artefactos de projectos em termos de pacotes:

- Organização por casos de utilização: Esta é a aproximação, por exemplo, do ICONIX (ver Capítulo 11) em que o sistema e respectivos modelos se encontram distribuídos por pacotes, consoante os vários casos de utilização.
- Organização por tipos de modelos: Esta é a aproximação, por exemplo do RUP (ver Capítulo 10), em que o sistema se encontra dividido por diferentes pacotes consoante as diferentes vistas, ou tipos de modelos produzidos. Há por exemplo, um pacote (com uma eventual hierarquia de pacotes) para o modelo de casos de utilização; outro para o modelo de análise; outro para o modelo de desenho; e outro ainda para o modelo de implementação.

Um aspecto correlacionado com o anterior, na definição e gestão dos pacotes e dos seus correspondentes artefactos é o esforço no sentido

da minimização das dependências entre si, assim como na minimização de dependências entre os respectivos artefactos.

4.9 Exercícios

Ex.24. Das seguintes afirmações assinale as que são verdadeiras:

- O UML é uma metodologia orientado por objectos.
- O UML é independente das ferramentas de modelação.
- O UML é um standard OMG.
- O UML é uma linguagem de programação robusta.

Ex.25. Quais são os dois aspectos importantes que se ganham com a adopção do UML.

Ex.26. Quais são os principais tipos de relações identificados na estrutura de conceitos do UML?

Ex.27. Com base em que princípio de modelação o UML propõe vários tipos de diagramas (com base nos quais se podem produzir visões complementares de um sistema)?

Ex.28. O que é uma marca com valor? Para que serve? Dê um exemplo de aplicação.

Ex.29. O que é um pacote UML? Enumere as três principais motivações/benefícios para a utilização de pacotes.

Capítulo 5 - UML – Casos de Utilização

Tópicos

- Introdução
- Casos de Utilização
- Diagramas de Casos de Utilização
- Proposta de Metodologia
- Exercícios

5.1 Introdução

A engenharia de requisitos [Sommerville97, Kulak00, Cockburn00] é uma área que se preocupa entre outros aspectos com a captura de requisitos de um sistema de software, o seu armazenamento e respectiva gestão.

Um **requisito** é uma especificação de uma determinada acção ou determinada condição que o sistema deverá satisfazer. Um **requisito funcional** descreve uma determinada acção (ou função) que o sistema deverá suportar. Por outro lado, um requisito não funcional descreve um aspecto (não funcional) que o sistema deverá satisfazer. **Requisitos não funcionais** têm a ver com aspectos gerais do sistema tais como: desempenho, robustez, fiabilidade, distribuição, segurança, integração com a Internet, abertura, ou suporte de standards.

Por influência de métodos desenvolvidos na primeira metade da década de 1990, em particular pelo método de Ivar Jacobson, OOSE [Jacob-

son92], o UML inclui os diagramas de casos de utilização (*use cases*) que permitem a especificação de requisitos funcionais segundo uma aproximação focada primordialmente nos utilizadores do sistema. A modelação de requisitos funcionais (e mesmo o próprio desenvolvimento de software) através de especificação de casos de utilização é actualmente considerada como uma abordagem extremamente adequada, quer por facilitar a comunicação entre a equipa de projecto e os clientes/utilizadores, quer ainda por promover a comunicação, gestão e condução no desenvolvimento do próprio projecto.

Esta relação entre casos de utilização e requisitos não é pacífica e clara. Antes pelo contrário. Há autores, por exemplo D. Kulak e E. Guiney [Kulak00] ou C. Larman [Larman99], que consideram que os casos de utilização são uma boa forma de representar os requisitos. Por outro lado, há outros autores, por exemplo D. Rosenberg [Rosenberg-99], que consideram que requisitos e casos de utilização são conceitos distintos, existindo todavia relações entre si, que deverão ser captadas.

Há alguns aspectos importantes na especificação de requisitos, que têm a ver com a sua apresentação, organização e nível de detalhe.

Quanto à sua apresentação e organização, há autores que advogam que os requisitos devem ser apresentados visualmente através de diagramas de casos de utilização e seguidamente cada caso em particular deverá ser especificado em detalhe através de uma descrição textual (segundo um determinado formato definido e usado consistentemente pela equipa de projecto) e, opcionalmente, através de um ou mais diagramas de colaboração e/ou desenhos de protótipos de interfaces homem-máquina (e.g., *screenshots* de protótipos de aplicações).

Quando os requisitos são descritos textualmente, recomenda-se normalmente que sejam numerados sequencialmente segundo, pelo menos, duas sequências distintas: uma para os requisitos funcionais e outra para os requisitos não funcionais. Por outro lado, quando os requisitos são baseados nos casos de utilização, deve usar-se o elemento pacote do UML como elemento estruturante.

A nossa opinião relativamente a estas duas posições é que qualquer uma é válida desde que seja assumida e usada consistentemente. Da

nossa experiência em projectos reais, concluímos que a aproximação de representar e organizar os requisitos através de casos de utilização facilita, em geral, a comunicação com os utilizadores e clientes e torna mais fácil e eficiente a sua captura e organização.

O aspecto do nível de detalhe da especificação de requisitos e casos de utilização tem a ver com o tipo de projecto em particular, o perfil e exigência do cliente, o tipo de sistema a especificar, etc. Consoante essas variáveis, é perfeitamente possível ter-se um caso de utilização típico descrito em 1 a 3 páginas ou em 10 a 30 páginas; e, claro, consequentemente é possível ter-se um sistema especificado numa dezena ou em algumas centenas de páginas. O compromisso da escolha de um nível de detalhe adequado nem sempre é fácil. Se é muito detalhado, o cliente acaba por responder "*é um bom trabalho, mas um pouco grande, talvez confuso, é capaz de ter algumas incoerências, …*" e claro que normalmente tem dificuldade de o "digerir" convenientemente. Se, por outro lado, é muito geral, o cliente responde qualquer coisa como "*sim, é isto que eu quero, mas não percebo muito bem o que irá fazer, nem as capacidades que irá, de facto, apresentar…*". A nossa experiência sugere que é importante uma reflexão, projecto a projecto, sobre o nível de detalhe da especificação de casos de utilização. Em geral, é preferível a descrição de casos de utilização de forma não muito detalhada, variando entre uma a cinco páginas.

5.2 Casos de Utilização

Conceito

Um **caso de utilização** é "uma sequência de acções que um ou mais actores realizam num sistema de modo a obterem um resultado parti-cular" [OMG99].

O modelo de casos de utilização permite capturar os requisitos de um sistema através do detalhe de todos os cenários que os utilizadores podem realizar. Os casos de utilização, mais que iniciar a modelação de requisitos de um sistema, dirigem/conduzem todo o processo de desenvolvimento.

Um caso de utilização é representado graficamente através de uma oval com uma frase que representa uma determinada acção. A frase deve ser escrita na voz activa, com um verbo no infinitivo. Por exemplo, “Submeter Proposta”, “Criar factura”, “Calibrar roda” ou “Validar utilizador” conforme ilustrado na Figura 5.1.

Figura 5.1: Representação gráfica de casos de utilização.

Um caso de utilização deve descrever o que faz um sistema (ou parte deste), mas não como é que tal é realizado. O foco é portanto na visão externa do sistema, ou seja, na visão que os utilizadores têm dele.

5.2.1 Casos de utilização e Cenários

Um **cenário** é uma determinada sequência de acções que ilustra um comportamento do sistema. Numa definição mais abstracta, deve-se entender um cenário como uma instância de um caso de utilização, sendo normal que um caso de utilização possa ser descrito por dezenas de possíveis cenários. Uma designação alternativa para cenário por vezes utilizada é “fluxo de acções”.

Deve-se especificar o comportamento de um caso de utilização descrevendo textualmente um ou mais fluxos de acções, de modo que um utilizador não técnico o possa entender sem dificuldade. Tal especificação deve incluir:

- Como e quando o caso de utilização começa e termina
- Quando é que o caso de utilização interactua com os actores
- Que objectos são trocados
- Cenário principal, e
- Cenários alternativos (e.g., situações de excepção)

Outras formas alternativas ou complementares, podem ainda incluir a especificação de pré e pós condições, os actores que iniciam o caso de utilização, os actores que beneficiam com o caso de utilização, um ou mais diagramas de interacção, etc.

Como exemplo apresenta-se de seguida a especificação textual do caso de utilização "Validar Utilizador" ilustrado na Figura 5.1. Considere-se que este caso de utilização pertence a um sistema tipo "caixa de multi-banco".

Exemplo 5.1: Especificação textual do caso de utilização "Validar Utilizador".

Nome: Validar Utilizador

Cenário Principal

O caso de utilização inicia-se quando o sistema apresenta um ecrã a pedir ao cliente o seu cartão electrónico. O cliente introduz o seu cartão MB e, através de um pequeno teclado, o seu PIN. Note-se que o cliente pode limpar a introdução do seu PIN inúmeras vezes e reintroduzir um novo número antes de activar o botão "Entrar". O cliente activa o botão "Entrar" para confirmar. O sistema lê o PIN e a respectiva identificação do cartão MB, e verifica se é válido. Se o PIN for válido o sistema aceita a entrada e o caso de utilização termina.

Cenário Alternativo 1 (Cliente cancela operação)

O cliente pode cancelar a transação em qualquer momento activando o botão "Cancelar", implicando a reinicialização do caso de utilização. Não é realizada qualquer alteração à conta do cliente.

Cenário Alternativo 2 (PIN inválido)

Se o cliente introduz um PIN inválido, o cartão MB é ejectado e o caso de utilização reinicializado. Se tal ocorrer 3 vezes consecutivas, o sistema cativa (i.e., "recolhe") o cartão MB e cancela a transação; não permitindo qualquer interacção nos 2 minutos seguintes.

Um aspecto importante na especificação do caso de utilização tem a ver com o nível de detalhe do mesmo. A dimensão da especificação de um caso de utilização varia significativamente consoante o tipo de projecto envolvido, os intervenientes do projecto, e as exigências impostas pelos clientes. Essa especificação normalmente é textual, de modo mais ou menos informal, mas pode ser complementada por diagramas de interacção (para ajudar a clarificar as interacções entre os diferentes intervenientes); ou até por protótipos de interfaces com utilizadores (e.g., desenho de ecrãs ou de listagens tipo).

Todavia, sempre que possível, deve-se evitar adoptar uma linguagem dependente da implementação e de aspectos tecnológicos. Designadamente, deve-se evitar referir explicitamente: pessoas, em vez de papéis desempenhados; departamentos específicos de uma organização; componentes de interface com o utilizador (e.g., botões, menus, caixas de texto, *scrollbars*); ou referências a dispositivos hardware. Deve-se igualmente evitar descrições em formatos técnicos ou formais, tais como pseudocódigo ou especificações em OCL [Warmer99], ou Z [Diller94].

5.2.2 Relações entre Casos de Utilização

Os casos de utilização podem encontrar-se relacionados através de três tipos de relações: generalização, inclusão e extensão. Estas relações potenciam significativamente a reutilização da especificação de requisitos. Ou seja, estas relações permitem ao analista aquilo que o programador de linguagens orientadas por objectos normalmente pratica: reutilizar trabalho já efectuado. Este é um aspecto essencial da filosofia dos casos de utilização e que normalmente não é facilmente apreendido pelos praticantes inexperientes.

Repare-se que o objectivo, neste contexto, não é a reutilização de código, mas sim a reutilização de especificações (normalmente textuais), com todos os benefícios daí decorrentes.

Generalização

Uma relação de **generalização entre casos de utilização** permite definir casos à custa de outros já existentes, pelo mecanismo de especi-

alização, ou alternativamente, permite definir casos mais abstractos a partir de casos concretos pelo mecanismo da redução ou generalização.

O caso de utilização filho herda o comportamento e semântica do seu pai, pode substituir especificações definidas no seu pai, bem como, pode introduzir novas especificações que lhe sejam específicas.

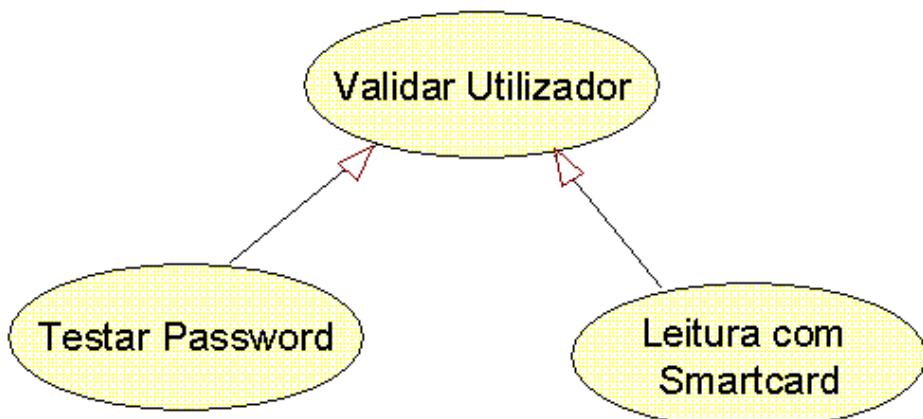



Figura 5.2: Relação de generalização entre casos de utilização.

A Figura 5.2 ilustra a relação de generalização entre casos de utilização. Na prática ilustra que o caso "Validar Utilizador" é especializado em outros dois, que utilizam diferentes mecanismos de identificação do utilizador: "Testar Password" e "Leitura com *Smartcard*".

Inclusão

A relação de **inclusão («include») entre casos de utilização** corresponde a uma relação típica de delegação, significando que o caso base incorpora o comportamento do outro caso relacionado. Usa-se a relação de inclusão para evitar a descrição dos mesmos fluxos de acções inúmeras vezes. A relação de inclusão é representada por uma relação de dependência (seta a tracejado) com o estereótipo «include».

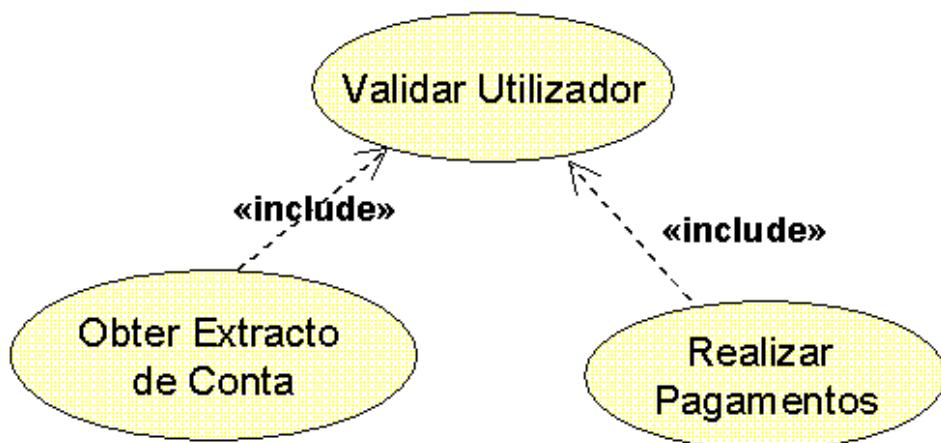



Figura 5.3: Relação de inclusão entre casos de utilização.

A Figura 5.3 ilustra um exemplo de utilização da relação de inclusão. Considere-se uma vez mais o exemplo da caixa de multibanco acima apresentado. Os casos de utilização “Obter Extracto de Conta” ou “Realizar Pagamentos” exigem que seja realizada previamente uma validação do respectivo utilizador. Para que essa funcionalidade não seja especificada mais que uma vez, os casos anteriores incorporam-na (como sua) ao estabelecerem uma relação de inclusão com o caso “Validar Utilizador”.

A questão importante que se coloca é como indicar, em que ponto da especificação, o caso de utilização relacionado deve ser incorporado no caso base. Este é um aspecto essencial na compreensão e domínio dos casos de utilização. A explicitação desta informação deve ser efectuada na especificação textual do caso de utilização. O Exemplo 5.2 clarifica esta questão com a descrição textual do caso “Obter Extracto” em que é evidente a indicação do ponto de inclusão através do texto “Incluir…”.

Exemplo 5.2: Especificação textual do caso de utilização “Obter Extracto de Conta”.

Nome: Obter Extracto de Conta

Cenário Principal

Incluir caso de utilização “Validar Utilizador”. Obter e verificar o número da conta.. Seleccionar todas as linhas de movimentos realizados nos

últimos 30 dias. Produzir uma lista resumo com esses movimentos, apresentando a data, o tipo de movimento (débito ou crédito), uma breve descrição e o valor do movimento. Produzir o saldo corrente da conta. Emitir um documento com essa informação, ejectando no terminal de Multibanco o referido documento. Apresentar mensagem no visor do terminal para o cliente retirar o extracto. Registar na conta do cliente que esta operação foi realizada com sucesso.

Cenário Alternativo 1

…

Extensão

Uma relação de **extensão («extend») entre casos de utilização** significa que o caso base incorpora implicitamente o seu comportamento num local especificado indirectamente pelo caso que é usado. Ou seja, o caso destino pode ser estendido com o comportamento de outro(s) caso(s). Uma relação de extensão permite representar:

- A parte de um caso que um utilizador vê como opcional, ou como existindo várias alternativas.
- Um subfluxo de acções que é executado apenas se determinadas condições se verificarem.
- Vários fluxos de acções que podem ser inseridos num determinado ponto de extensão, de forma controlada, através de uma interacção explícita com um actor.

O caso de utilização destino é estendido num ou mais pontos, designados por **pontos de extensão** os quais são mecanismos de variabilidade [Jacobson97]. Ou seja, são pontos de entrada do caso de utilização que lhe dá algum nível de configurabilidade e versatilidade.

(Nota: Os pontos de extensão são um dos mecanismos de variabilidade mais comuns no desenho de sistemas de componentes, de *frameworks* aplicacionais ou de *frameworks* de *middleware,* que exigem exactamente um elevado grau de versatilidade.)

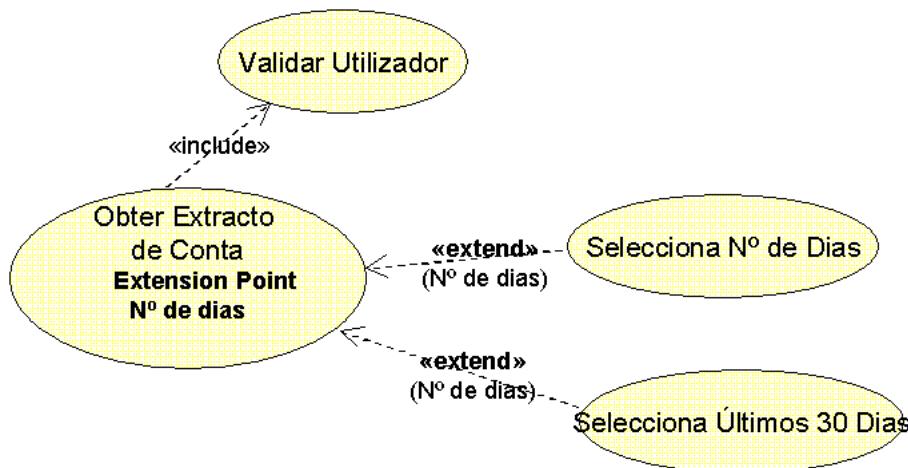



Figura 5.4: Relação de extensão entre casos de utilização.

Considere-se o caso "Obter Extracto de Conta" do exemplo anterior. Na descrição textual do Exemplo 5.2, a especificação do número de dias respeitantes à selecção dos movimentos a visualizar era estática (30 dias). Numa situação mais flexível, o ideal é que o cliente/utilizador pudesse seleccionar o número de dias pretendido, ou simplesmente activar um botão para indicar que pretendia seleccionar os movimentos relativos aos últimos 30 dias.

Figura 5.4 ilustra esta situação através da utilização da relação de extensão, representada por uma relação de dependência (seta a tracejado) com o estereótipo «extend». Pelo facto do caso de utilização destino poder ter vários pontos de extensão, especifica-se, na relação de extensão, qual o ponto de extensão a que diz respeito (neste caso "(N.º de dias)").

O Exemplo 5.3 ilustra a forma de representação textual dos pontos de extensão nos casos de utilização, bem como, a especificação textual de um caso de utilização que permite estender outro num determinado ponto de extensão.

Exemplo 5.3: Especificação textual do caso de utilização "Obter Extracto de Conta" revisto.

Nome: Obter Extracto de Conta

Pontos de Extensão:

N.º de dias

Cenário Principal

Incluir caso de utilização "Validar Utilizador". Obter e verificar o número da conta. Seleccionar o n.º de dias com base no qual se produz o extracto. (N.º de dias). Por omissão são seleccionados os últimos 30 dias. Produzir uma lista resumo com esses movimentos, apresentando a data, o tipo de movimento (débito ou crédito), uma breve descrição e o valor do movimento. Produzir o saldo corrente da conta. Emitir um documento com essa informação produzida ejectando no terminal de Multibanco o referido documento. Apresentar mensagem no visor do terminal para o cliente retirar o extracto. Registar na conta do cliente que esta operação foi realizada com sucesso.

Cenário Alternativo 1

…

Exemplo 5.4: Especificação textual do caso de utilização "Selecciona N.º de Dias".

Nome: Selecciona N.º de Dias

Tipo: Abstracto

Cenário Principal

É apresentado um ecrã em que o utilizador pode especificar o n.º de dias desejado, através da marcação em vários botões numéricos (de '0' a '9'). Há uma caixa de texto construída dinamicamente que vai apresentando o valor corrente. Por fim, o utilizador marca o botão "Confirmar" e o valor construído é retornado ao caso destino no seu respectivo ponto de extensão.

Cenário Alternativo 1

Idêntico ao cenário principal. Em qualquer momento o utilizador pode marcar sobre o botão "Apagar" de modo a apagar o algarismo introduzido mais recentemente.

Cenário Alternativo 2

Idêntico ao cenário principal. Quando o utilizador marca "Confirmar" e o valor introduzido for superior a 59 dias é apresentada uma mensagem de aviso que o número máximo é 59, e o caso é reiniciado.

Cenário Alternativo 3

Idêntico ao cenário principal. Em qualquer momento o utilizador pode seleccionar o botão "Cancelar"– O caso termina e é retornado o valor 1 (dia) por omissão.

Em situações reais este tipo de relações pode-se complicar significativamente. Imagine-se por exemplo, que um outro ponto de extensão é a língua escolhida que terá como consequência que todas as interfaces com o utilizador sejam compatíveis com a língua respectiva.

5.3 Diagramas de Casos de Utilização

Um **diagrama de casos de utilização** ilustra um conjunto de casos de utilização, de actores e suas relações (ver Figura 5.5). As suas aplicações comuns são:

- Para modelar o contexto de um sistema. Neste caso, a ênfase encontra-se na identificação da fronteira do sistema, dos seus actores e no significado das suas funções.
- Para modelar os requisitos de um sistema. Consiste na identificação do que o sistema deve fazer, independentemente de como o sistema o deve realizar.

A ligação entre um actor e um caso de utilização corresponde a uma relação de comunicação (de estereótipo «communicate») entre estes dois elementos. É representada por uma linha a cheio e em geral sem setas, pois a comunicação entre o actor e o caso pode ser em ambos os sentidos, ou pode nem sequer ser relevante (ou possível) determinar essa informação.

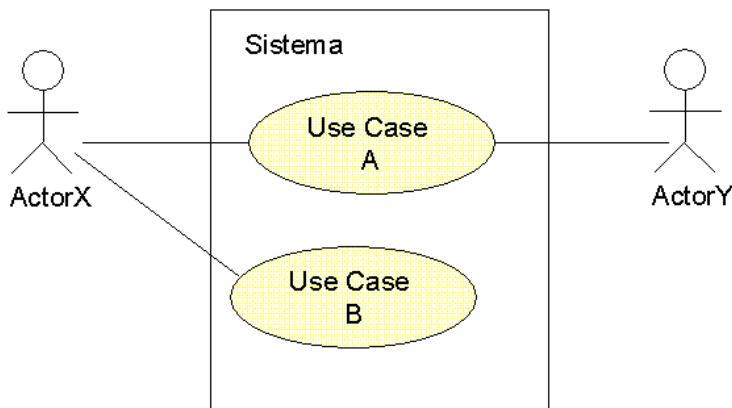



Figura 5.5: Diagrama de casos de utilização.

5.3.1 Actores

Um **actor** é o conceito que representa, em geral, um papel que um utilizador desempenha relativamente ao sistema em análise. Todavia, um actor não é necessariamente um papel de um utilizador; pode cor-

responder a um papel desempenhado por um outro sistema informático, por um equipamento hardware especializado, ou pela simples passagem de tempo. O conjunto total de actores de todos os casos de utilização reflecte todos os elementos que interactuam com o sistema.

O ícone do estereótipo actor representa por omissão a figura de um indivíduo, mas podem-se definir outros estereótipos (com respectivos ícones) para diferentes tipos de actores.

Um determinado utilizador pode desempenhar diferentes papéis, podendo, por conseguinte, representar diferentes actores. Por exemplo, na situação da caixa Multibanco, poder-se-ão identificar pelo menos dois actores: o cliente do banco, que acede ao sistema para realizar variadas operações bancárias; e o operador da agência ou da caixa, que é responsável pela sua activação, por carregar dinheiro na máquina, etc.

Os actores podem encontrar-se relacionados através de relações de generalização, conforme ilustrado na Figura 5.6, o que significa que o actor-filho (na relação de generalização) herda todas as funcionalidades e todos os papéis do seu actor-pai, podendo adicionalmente apresentar as suas próprias funcionalidades.

Figura 5.6: Relação de hierarquia entre actores.

5.3.2 Casos de Utilização Abstractos e Concretos

Por definição, um **caso de utilização abstracto** é um caso que não apresente uma relação de comunicação com qualquer actor. Por outro lado, um **caso de utilização concreto** é um caso que não é abstracto.

O nome dos casos abstractos é representado a itálico de forma a distingui-los dos casos concretos (ver Figuras 5.8 e 5.9).

Este tipo de casos de utilização são tipicamente casos envolvidos em relações de generalização (e.g., o caso pai), inclusão (o caso destino) ou de extensão (o caso base) com outros casos de utilização. O objectivo destes casos abstractos é dar ao modelo um nível elevado de reutilização e de flexibilidade, como se viu na Secção 5.2.

5.4 Proposta de Metodologia

Apesar da estrutura de conceitos associada aos diagramas de casos de utilização ser relativamente simples, como se viu nas secções anteriores, a prática vem mostrar que a sua aplicação não é trivial, havendo inúmeras situações de utilização incorrecta. Propomos nesta secção uma metodologia para desenho de diagramas de casos de utilização de forma a ajudar o leitor nesta actividade "aparentemente" fácil de realizar.

Para facilitar a discussão, considere-se o exemplo de um sistema "Máquina de Bebidas". Uma máquina de bebidas é um sistema colocado normalmente em locais estratégicos, por onde passa muita gente, como sejam paragens de metro, recintos desportivos, escolas, etc.

A metodologia proposta consiste na aplicação sucessiva dos seguintes passos:

1) Identificar os actores do sistema, ou seja, o perfil de indivíduos e de outros sistemas que interactuam com o sistema original.

2) Identificar, para cada actor, os seus casos de utilização principais. Note-se que podem existir casos que envolvam a participação de mais que um actor.

3) Com base nos casos de utilização originais, identificar, factorizar e colocar em evidência casos de utilização que sejam recorrentes em mais que um dos casos originais. Nessa situação, cria-se o novo caso de utilização (em geral é um caso abstracto) e os casos originais envolvidos estabelecem uma relação de inclusão com o dito caso. Repetir o processo até não se conseguir identificar qualquer outro caso a reutilizar.

4) Para tratar casos de utilização que pretendam ser flexíveis e versáteis, definir pontos de extensão (ou de variabilidade) e conjuntamente definir um ou mais casos de utilização (abstractos) que os permitam estender nesses pontos. Nesta situação, cria-se uma relação de extensão do caso abstracto para o caso estendido.
5) Especificar textualmente cada caso de utilização segundo um determinado formato previamente definido. Não esquecer nesta especificação textual a explicitação dos pontos de extensão e de inclusão anteriormente identificados.

Tendo em conta a metodologia proposta, o passo 1 começa com a identificação dos actores do sistema bem como das suas principais actividades. Identificamos três actores: (1) o cliente, que compra a bebida; (2) o agente do fornecedor, que é responsável por carregar as bebidas na máquina; e (3) o dono da máquina, que é responsável por retirar o dinheiro da máquina.

No passo 2 foram identificados os principais casos. A Figura 5.7 ilustra a versão preliminar do respectivo diagrama de casos de utilização. Neste passo é crucial a escolha do nível de granularidade adequado para "captar" os casos envolvidos. Recorde-se a definição de caso de utilização para se realizar essa escolha. Note-se que um caso não é simplesmente "uma acção", ou mesmo "uma sequência de acções", mas sim "uma sequência de acções que um ou mais actores realizam num sistema de modo a obterem um resultado particular".

Note-se que o caso "Repor Bebidas" é realizado conjuntamente pelo agente do fornecedor de bebidas e pelo dono da máquina, este último responsável por abrir a máquina e supervisionar o processo envolvido.

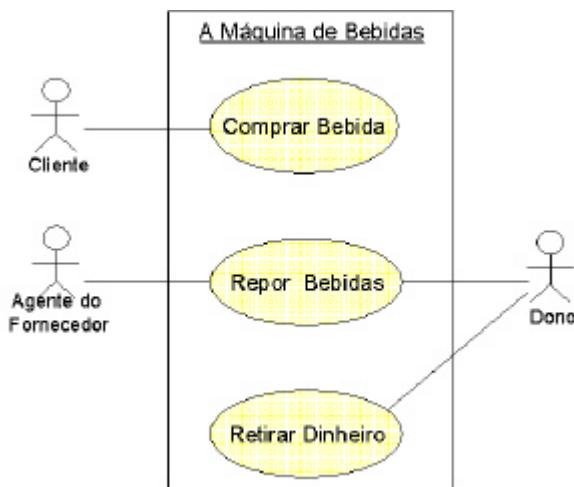
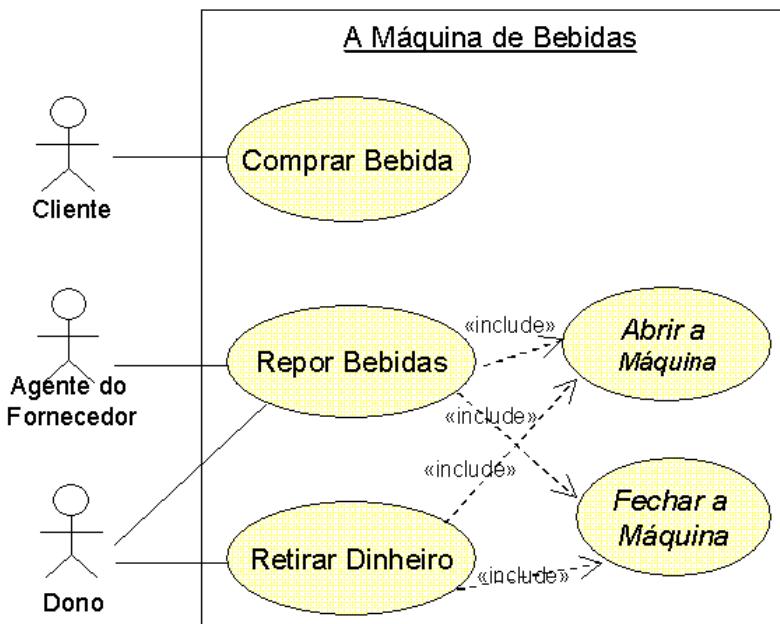



Figura 5.7: Diagrama de casos de utilização da "Máquina de Bebidas" – preliminar.

Passando para o passo 3 da metodologia acima enunciada, deve-se identificar comportamentos comuns realizados por mais que um dos casos do sistema. Neste exemplo, constata-se que os casos "Repor Bebidas" e "Retirar Dinheiro" envolvem dois tipos de acções comuns "Abrir Máquina" e "Fechar Máquina". Este facto deve ser factorizado através de dois casos de utilização correspondentes e devem ser estabelecidas as respectivas relações de inclusão (ver Figura 5.8).

Figura 5.8: Diagrama de casos de utilização da “Máquina de Bebidas” – com inclusão.

O passo 4 da metodologia pressupõe que a análise dos casos de utilização existentes e que seja avaliada a necessidade da criação de pontos de extensão para um ou mais casos. No exemplo, vamos considerar que o caso “Repor Bebidas” tem um ponto de extensão designado por “encher prateleira”, que permite associar ao caso de utilização um ou mais casos abstractos que providenciem diferentes algoritmos para a reposição de bebidas nas prateleiras.

Figura 5.9: Diagrama de casos de utilização da "Máquina de Bebidas" – com extensão

A Figura 5.9 ilustra essa situação com o caso "Repor Bebidas de Acordo com as Vendas", a reposição de bebidas na máquina tem em conta o número de bebidas vendidas, implicando que se colocam mais latas das bebidas mais vendidas. Note-se que poder-se-iam definir outros casos de extensão abstractos, que implementavam diferentes algoritmos ou estratégias de reposição de bebidas; por exemplo: repor bebidas de forma uniforme (o mesmo número de latas, por tipo de bebida); repor bebidas de acordo com a marca; etc.

5.5 Exercícios

Ex.30. Indique 2 vantagens da visualização de um caso de utilização.

Ex.31. Com base no exemplo da "Máquina de Bebidas" descrito na Secção 5.4 complete a descrição dos requisitos do sistema ao especificar textualmente os casos de utilização definidos (passo 5 da metodologia proposta).

Ex.32. Esboce um diagrama de casos de utilização para um controlo remoto de TV. Garanta que inclui todas as funções do controlo remoto como casos de utilização do seu modelo. Descreva textualmente os casos de utilização "Ligar TV" e "Seleccionar Canal". Sugestão: Considere que a TV tem um sistema de *password*, configurado opcionalmente, para que os pais tenham a garantia que os filhos não passem muitas horas em frente ao televisor!

Ex.33. Analise os processos RUP e ICONIX e discuta as suas respectivas interpretações relativamente aos conceitos "requisitos" e "casos de utilização".

Ex.34. Discuta as vantagens/desvantagens da aplicação de diagramas de casos de utilização na produção de cadernos de encargo e/ou propostas de sistemas de software.

Ex.35. Discuta as vantagens/desvantagens da adopção de um estilo de escrita dos casos de utilização na óptica dos seus utilizadores. Sugestão: considere a possibilidade de geração de documentação.

Ex.36. Considere o sistema de uma equipa de futebol constituído pelos seguintes actores: jogador, treinador, atacante, guarda-redes, médio, defesa, presidente. Desenhe o respectivo diagrama de casos de utilização. Sugestão: considere por exemplo os seguintes casos: jogar, treinar, defender a baliza, pagar ao jogador, pagar ao treinador, vender jogador, contratar jogador, contratar treinador, despedir treinador.

Ex.37. Faça um diagrama de casos de utilização a partir do manual de utilizador de uma determinada aplicação. Considere por exemplo o Word da Microsoft ou outra qualquer aplicação do seu conhecimento.

Capítulo 6 - UML – Modelação da Estrutura

Tópicos

- Introdução
- Classes
- Relações
- Interfaces
- Instâncias e Objectos
- Diagramas de Classes e Diagramas de Objectos
- Exemplos e Recomendações
- Exercícios

6.1 Introdução

A modelação da estrutura de um sistema de software consiste principalmente, segundo a abordagem orientada por objectos, na identificação de classes e suas respectivas relações.

Um objecto reflecte em geral uma entidade do mundo real e apresenta um estado e comportamento próprio. Os objectos interactuam entre si por troca de mensagens. Uma classe consiste numa estrutura que permite criar objectos semelhantes; i.e., que apresentem estado e comportamento semelhante. Neste sentido diz-se que uma classe é uma fábrica de objectos e que um objecto é uma instância de uma classe.

O UML providencia os seguintes elementos, que permitem a especificação da estrutura ou estática de um sistema de software: classes, relações, interfaces, objectos. Com base nestes elementos podem-se definir dois tipos de diagramas com fins complementares: diagramas de classes e diagramas de objectos, que são o foco dos conceitos em análise neste capítulo.

(Consulte-se o Capítulo 3 e em particular a Secção 3.5 para mais detalhes sobre estes conceitos e uma discussão geral sobre as metodologias orientadas por objectos.)

6.2 Classes

Uma **classe** é a descrição de um conjunto de objectos que partilham os mesmos atributos, operações, relações e a mesma semântica. Uma classe corresponde a algo tangível ou a uma abstracção conceptual existente no domínio do utilizador ou no domínio do engenheiro de software.

Uma classe bem estruturada é simples e facilmente entendida; providencia uma abstracção definida a partir do vocabulário do domínio do problema ou do domínio da solução; agrega um conjunto restrito e bem definido de responsabilidades; e providencia uma separação clara entre a especificação abstracta e a sua implementação.

Uma classe é representada em UML por um rectângulo (ver Figura 6.1) com uma, duas ou três secções. Na primeira secção apresenta-se o nome da classe, na segunda a sua lista de atributos, e na terceira a sua lista de métodos. Pode ainda ter, opcionalmente, uma quarta secção, onde se poderá especificar outra informação (e.g., a lista de responsabilidades que a classe assume).

O nome de uma classe (tal como de qualquer outro elemento do UML) pode ser apresentado na forma simples ou na sua forma completa. Nesta última situação, o nome do elemento é precedido pelo(s) nome(s) do(s) pacote(s) onde se encontra definido, separado pela *string* "**::**".

Figura 6.1: Representação de uma classe em UML.

Nas segundas e terceiras secções do ícone de classe apresentam-se respectivamente os seus atributos e os seus métodos, que podem apresentar-se com maior ou menor detalhe (ver Figura 6.2).

Na definição de atributos podem-se visualizar apenas os seus nomes, ou adicionalmente os respectivos tipos, ou ainda a visibilidade (e.g., se é um atributo público, privado, protegido) ou outras qualificações (e.g., se é variável estática). A definição completa de um atributo é expressa por:

visibility name [multiplicity] : type-expression = initial-value { property-string }

A *multiplicity* permite especificar a multiplicidade de um atributo. Por omissão assume-se multiplicidade 1..1 (exactamente um). O *initial-value* permite especificar o valor inicial do atributo, i.e., o valor que o atributo contém no momento da sua criação. Também é um parâmetro opcional (bem como o sinal “=” que o precede). O *property-string* especifica valores de propriedades que se podem aplicar ao elemento. Também é um parâmetro opcional (bem como os parêntesis-chavetas que o envolvem).

Na definição de operações (ou métodos) podem-se visualizar apenas os seus nomes, ou adicionalmente as respectivas assinaturas, ou ainda visibilidade (e.g., se é operação pública, privada, protegida) ou outras qualificações (e.g., se é abstracta ou polimórfica). A definição completa de uma operação é expressa por:

visibility name (parameter-list) : return-type-expression { property-string }

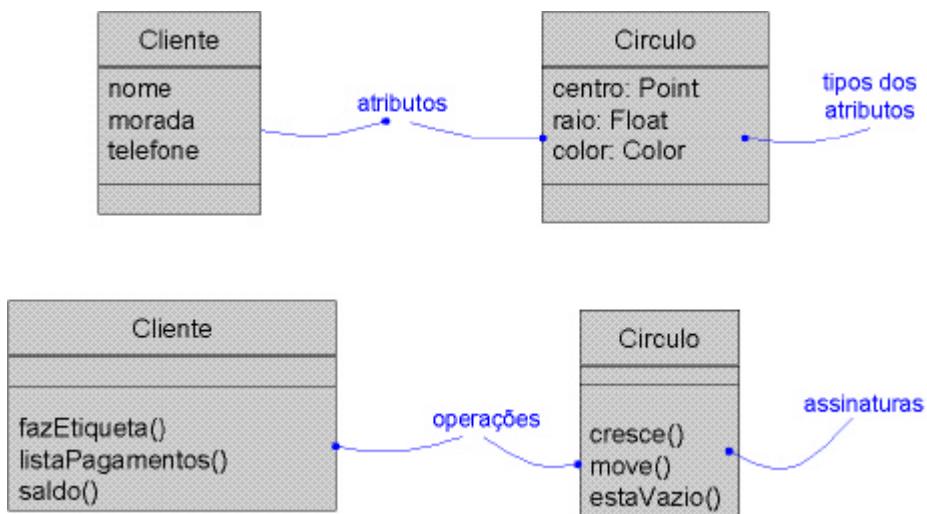



Figura 6.2: Apresentação dos atributos e operações de uma classe.

Podem-se definir subsecções dentro da segunda ou terceira secção de forma a melhor organizar e manter os atributos e operações de uma classe. Tal é realizado com base na noção de estereótipo conforme ilustrado na Figura 6.3.

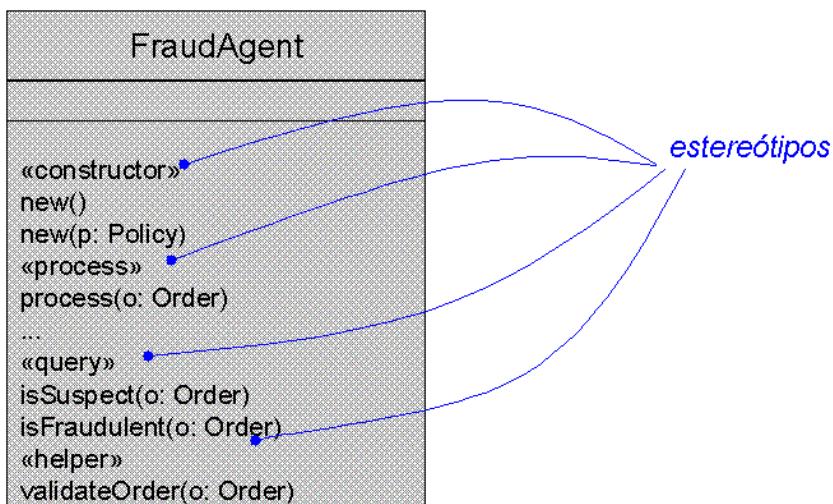



Figura 6.3: Usando estereótipos para organizar a lista de operações de uma classe.

6.3 Relações

Uma **relação** em UML estabelece a ligação entre elementos e é representada graficamente por um determinado tipo de linha. Na modelação orientada por objectos os três tipos de relações mais importantes são (1) dependências; (2) generalizações; e (3) associações, que de seguida se descrevem.

6.3.1 Relação de Dependência

Uma **relação de dependência**, ou simplesmente dependência, indica que a alteração na especificação de um elemento pode afectar outro elemento que a usa, mas não necessariamente o oposto. A dependência é representada em UML através de uma linha dirigida a tracejado.

No contexto de classes, usam-se dependências para ilustrar que uma classe usa outra classe como argumento na assinatura de uma das suas operações ou como tipo na definição dos seus atributos. Por motivos de simplicidade e clareza não se explícita em geral este tipo de relações nos diagramas de classes, já que esse tipo de dependência encontra-se especificado implicitamente.

Contudo, em UML as dependências são usadas, entre outros elementos, de modo mais pertinente, nomeadamente com elementos do tipo pacotes e notas.

A Figura 6.4 ilustra um exemplo da relação de dependência entre as classes `SensorTemperatura` e `Temperatura`.

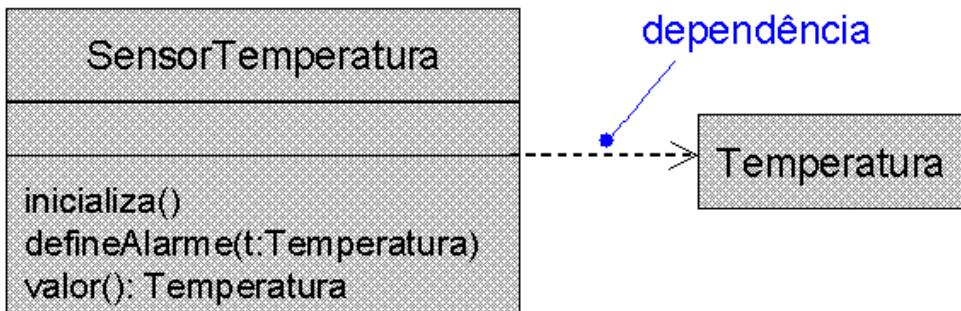



Figura 6.4: Dependência entre classes.

6.3.2 Relação de Generalização

Uma **relação de generalização**, ou simplesmente generalização, é uma relação entre um elemento geral (e.g., superclasse, super-caso-utilização, super-pacote) e um elemento mais específico (e.g., sub-classe, sub-caso-utilização, sub-pacote). Geralmente conhecida como uma relação do tipo "*is-a*" ou "*is-a-kind-of*" é representada em UML por uma linha dirigida a cheio com um triângulo a branco num seu extremo.

No contexto de classes usam-se generalizações para ilustrar as relações de herança conhecidas das linguagens de programação orientadas por objectos.

A herança providencia um mecanismo natural e potente de organização dos programas de software ao permitir: (1) que cada subclasse herde o estado e comportamento de uma superclasse; (2) subclasses podem adicionar o seu próprio estado e comportamento; e (3) as subclasses podem ainda alterar os métodos (i.e., comportamento) herdados, providenciando implementações especializadas desses métodos.

Os benefícios conhecidos da herança têm a ver com (1) possibilidade de reutilização do código definido na superclasse numa ou mais subclasses; e (2) definição de *frameworks* (programas com estruturas quase-completas) através de classes abstractas que definem comportamentos genéricos e/ou estilos de desenho comuns.

A Figura 6.5 exemplifica a aplicação da relação de generalização entre classes num sistema de representação de figuras geométricas. A classe `FiguraGeométrica` é especializada em duas hierarquias distintas, mas não disjuntas, de subclasses. Por um lado, conforme a dimensão "forma", tem-se as subclasses `Rectângulo`, `Círculo` e `Polígono` (a classe `Rectângulo` é por sua vez uma generalização da classe `Quadrado`). Por outro lado, conforme a adopção ou não de etiqueta, tem-se as subclasses `ComEtiqueta` e `SemEtiqueta`. Por fim, tem-se a subclasse Círculo`ComEtiqueta` que é uma combinação entre as duas dimensões.

Note-se neste exemplo que a `FiguraGeométrica` é uma classe abstracta (por isso, o seu nome é apresentado a *itálico*) que define o comportamento geral de todas as suas subclasses.

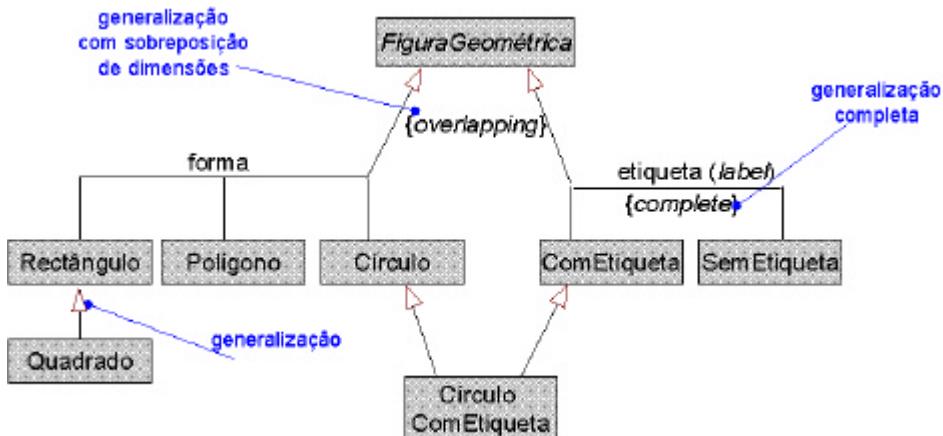



Figura 6.5: Relação de generalização entre classes.

Por omissão uma relação de generalização representa uma decomposição disjunta ou exclusiva. No entanto outras semânticas podem ocorrer, caso sejam especificadas através das seguintes restrições:

- **Generalização disjunta** `{disjoint}`: o tipo de generalização por omissão, que especifica o facto de uma classe descendente de X poder ser apenas descendente de uma das subclasses de X.
- **Generalização sobreposta** `{overlapping}`: especifica o facto de uma classe descendente de X pertence ao produto cartesiano das subclasses de X (no exemplo da Figura 6.5, a classe `CírculoComEtiqueta` é uma combinação das subclasses de `FiguraGeométrica`).
- **Generalização completa** `{complete}`: vs. **incompleta** `{incomplete}`: especifica o facto de uma generalização ser completamente especificada (o caso da decomposição segundo a dimensão "etiqueta" no exemplo da Figura 6.5) ou não.

6.3.3 Relação de Associação

Uma **relação de associação**, ou simplesmente associação, é uma relação estrutural que especifica que objectos de uma classe estão ligados a objectos de outra.

A Figura 6.6 ilustra a associação de "posse" entre as classes `Utilizador` e `Password`, com uma multiplicidade de "1 para muitos" (1-N). A associação indica que um utilizador tem várias (0 ou mais) *passwords* e que uma *password* pertence necessariamente a um utilizador.

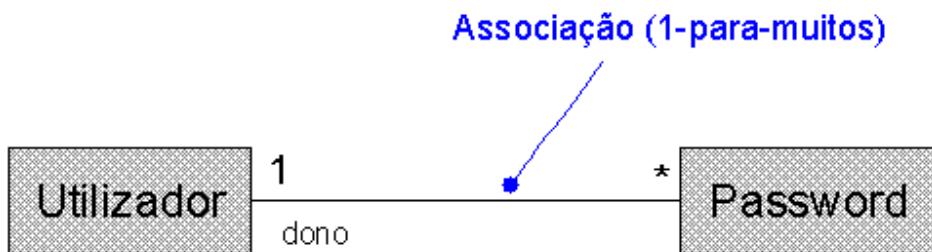



Figura 6.6: Relação de associação entre classes.

Uma associação é representada em UML por uma linha a cheio complementada por um conjunto de adornos que especificam diferentes informações (ver Figura 6.7), tais como:

- O nome;
- O papel de cada participante na associação;
- A multiplicidade de cada participante na associação;
- O tipo de agregação.

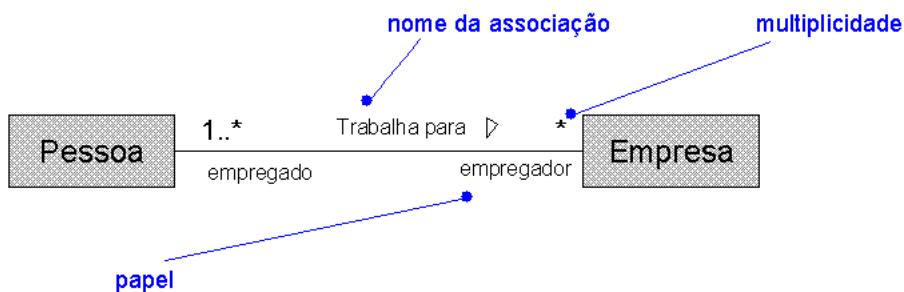



Figura 6.7: Adornos comuns usados em relações de associação.

As associações podem ainda incluir outros adornos, cuja utilização é em geral menos comum: navegação, visibilidade e qualificação.

Multiplicidade

A **multiplicidade** traduz o número de instâncias de uma classe que se podem relacionar (através da associação) com uma única instância da(s) outra(s) classe(s) participante(s). Pode-se especificar em UML qualquer tipo de multiplicidade. Por exemplo, multiplicidade muitos (`*`), um ou mais (`1..*`), exactamente um (`1`), zero ou um (`0..1`), um determinado número (e.g., 3), uma determinada gama (e.g., `2..6`), ou mesmo uma multiplicidade mais complexa especificada através de listas (e.g., `0..3, 5..7, 10..*` para representar "qualquer número de objectos excepto 4, 8 ou 9").

Navegação

A **navegação** traduz a forma como a partir de uma instância de uma classe se pode aceder a uma ou mais instâncias de outra classe relacionada pela associação. Por omissão a navegação numa associação é bidireccional. Contudo, há situações, em particular já na fase de desenho, que o que se pretende é representar uma associação unidireccional.

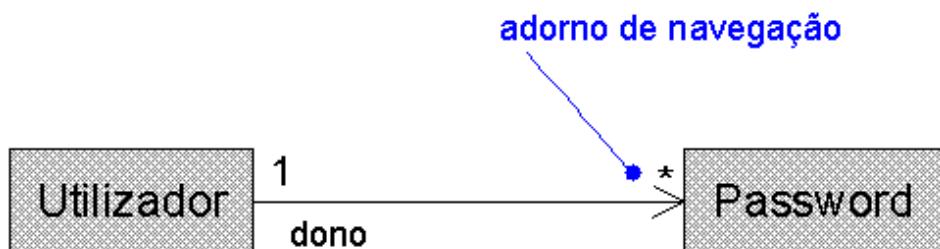



Figura 6.8: Navegação numa relação de associação.

Considere-se o diagrama da Figura 6.8 e que na fase de desenho chegou-se à conclusão da seguinte situação: "*Dado um utilizador consegue-se obter todas as respectivas instâncias de password. Mas dado uma password não é possível (ou relevante) obter-se o respectivo utilizador.*" A Figura 6.8 ilustra a utilização do adorno de navegação para tratar esta situação.

Agregação (Simples)

A associação entre classes sem agregação reflecte que ambas as classes se encontram no mesmo nível conceptual. Por outro lado, uma **relação de associação com agregação** traduz que existe uma relação do tipo "*is-part-of*" ou "*has-a*", o que corresponde ao facto de uma instância de determinada classe possuir ou ser composta por várias instâncias de outra classe. O adorno de agregação é representado por um losango colocado junto à classe que representa o elemento agregador ou "o todo".

A associação de agregação traduz apenas o facto de uma classe ser composta por diferentes outras classes, suas componentes.

A Figura 6.9 ilustra a relação de agregação entre várias classes. Na prática a descrição das diferentes componentes que compõem um computador pessoal (PC).

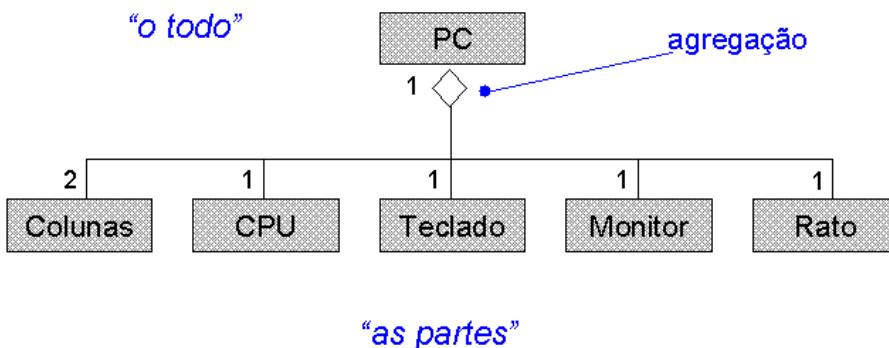



Figura 6.9: Agregação simples numa relação de associação.

Composição (Agregação Composta)

A **composição**, ou agregação composta, é uma variante à agregação simples, em que é adicionada a seguinte semântica: (1) forte pertença do "todo" em relação à "parte", e (2) tempo de vida delimitado (as "partes" não podem existir sem o "todo"). Adicionalmente, o "todo" é responsável pela disposição das suas "partes", ou seja, "o todo" é responsável pela criação e destruição das suas "partes".

O adorno de agregação composta é representado por um losango a cheio colocado junto à classe que representa o elemento agregador ou "o todo".

Figura 6.10: Agregação composta numa relação de associação.

A Figura 6.10 ilustra um exemplo de uma associação com agregação composta, de forma a reflectir o facto que "*um Departamento não existe fora do contexto de uma Empresa*".

Associações Qualificadas

Um **qualificador** é um atributo, ou lista de atributos, cujos valores servem para partir o conjunto de instâncias associadas a uma instância ao longo de uma associação. Os qualificadores são atributos da associação [OMG99].

Um qualificador é representado graficamente por um pequeno rectângulo junto de um extremo de uma associação. O rectângulo qualificador faz parte da associação e não do qualificador(es) que contem. O qualificador é colocado no extremo (da classe) origem da associação. Uma instância da classe origem, conjuntamente com um valor do qualificador, permite seleccionar univocamente um subconjunto das instâncias da classe destino, i.e. da classe do outro extremo da associação.

A multiplicidade afecta ao extremo destino denota a cardinalidade do conjunto das instâncias da classe destino, com base no par de informação: instância de origem e valor do qualificador. Os valores comuns são:

- “0..1”: um único valor pode ser seleccionado ou eventualmente nenhum.
- “1”: um único valor tem de ser seleccionado.
- “*”: o valor do qualificador é um índice que agrega as instâncias destino em diferentes subconjuntos.

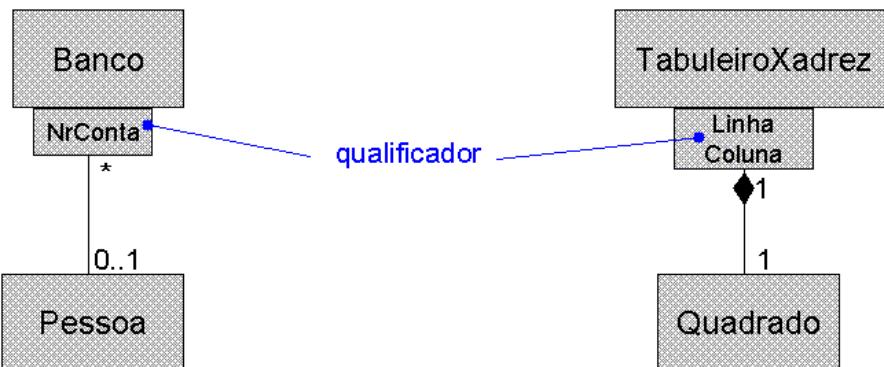



Figura 6.11: Associação com qualificador.

A Figura 6.11 ilustra dois exemplos de associações com qualificador entre as classes `Banco` e `Pessoa` e entre `TabuleiroXadrez` e `Quadrado`.

Associações Reflexivas

Uma associação diz-se **reflexiva** quando estabelece uma relação estrutural consigo própria. Este tipo de associação acontece quando uma classe tem objectos que desempenham diferentes papéis. Por exemplo, um ocupante de um carro pode desempenhar o papel de condutor ou de passageiro (ver Figura 6.12).

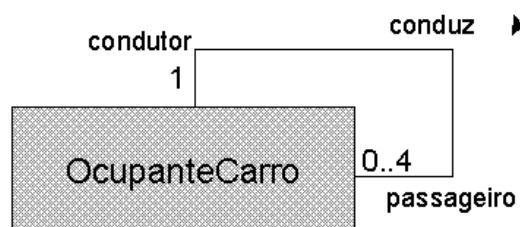



Figura 6.12: Associação reflexiva.

Visualmente é fácil identificar associações reflexivas pelo facto de corresponderem a linhas que têm origem e destino na mesma classe.

Classes-Associação

Numa relação de associação entre classes, a associação pode também ter os seus próprios atributos (e eventualmente operações), devendo ser, por conseguinte, modelizada também como uma classe. Este tipo de classes designa-se por **classe-associação**.

Considere-se o exemplo da Figura 6.13, em que a associação entre as classes `Pessoa` e `Empresa` traduz as tarefas que cada empregado realiza na empresa. Para cada tarefa é mantido um conjunto de atributos. A classe-associação `Tarefa` é representada visualmente como qualquer outra classe, mas apresenta uma linha a tracejado a ligá-la à linha da associação.

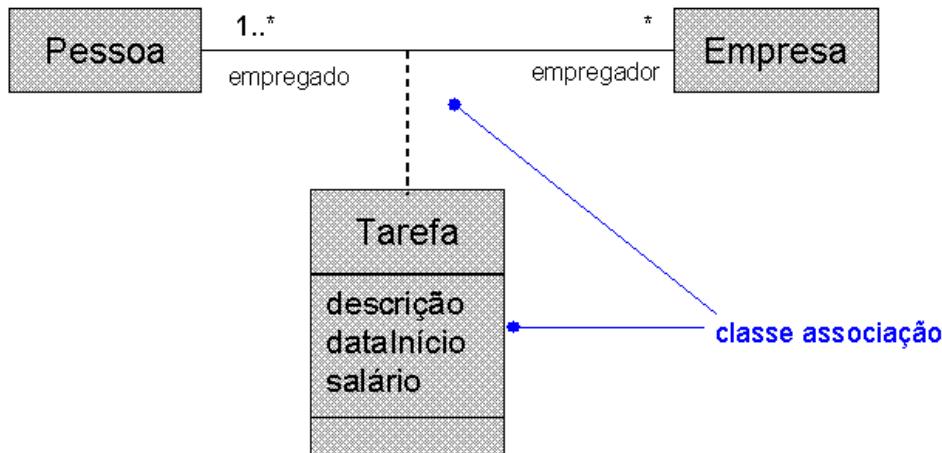



Figura 6.13: Exemplo de uma classe-associação.

Associações N-Árias (N ≥ 3)

Associações N-árias, com aridade maior ou igual a 3, são pouco comuns na modelação de classes. Contudo, há situações em que a aplicação deste tipo de associações é vantajosa em termos da clareza

do modelo. Nestas circunstâncias, a associação é representada por um losango com linhas para todas as suas classes participantes.

A multiplicidade em associações n-árias pode ser especificada, mas é menos óbvia que a multiplicidade em associações binárias. A multiplicidade num papel representa o número de tuplos (de instâncias) numa associação quando os outro N-1 valores são fixos.

A Figura 6.14 ilustra um exemplo de uma associação n-ária, a associação `Tarefa` e correspondente classe-associação, que relaciona as classes `Pessoa`, `Empresa` e `TipoTarefa`. Caso a associação tenha também atributos e/ou operações próprias, cria-se uma classe-associação, a qual é ligada ao losango por uma linha a tracejado, conforme ilustrado na Figura 6.14.

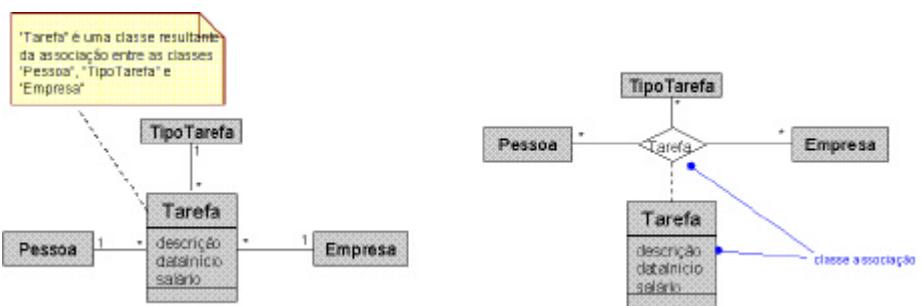



Figura 6.14: Associação ternária e uma classe-associação.

As associações N-árias podem geralmente ser transformadas em várias relações binárias entre a classe-associação e as restantes classes participantes. No entanto, se for esta a estratégia adoptada deve ser assinalado esse facto (por exemplo, através de um estereótipo ou de uma anotação) junto à classe-associação (ver Figura 6.14).

6.4 Interfaces

Uma **interface** é um contrato na forma de uma colecção de especificações de métodos que providencia um mecanismo para separação clara entre a vista externa e a vista interna de um determinado elemento (ver Figura 6.15).

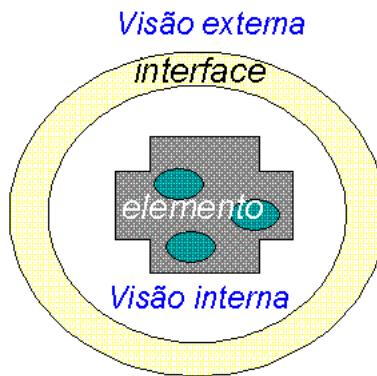



Figura 6.15: Noção de interface.

As interfaces permitem dar a conhecer um determinado elemento, escondendo os seus detalhes internos, por exemplo, os detalhes de implementação. Uma interface é realizada (ou implementada) por uma ou mais classes, as quais prometem implementar todos os métodos nela especificados. Note-se que em termos de álgebra computacional, tanto as classes como as interfaces são consideradas ao mesmo nível como “tipos”.

Em termos gerais, o conceito de interface apresenta os seguintes benefícios ao nível de programação:

- Captura de semelhanças entre classes não relacionadas sem forçar a criação de relações artificiais entre elas.
- Declaração de métodos que uma ou mais classes esperam implementar.
- Revelar a interface de programação de um objecto sem revelar a sua classe. Ou seja, um objecto pode ser visto de diferentes perspectivas (i.e., diferentes tipos) consoante as situações.

O conceito de interface é um mecanismo usual nas actuais linguagens de programação baseadas em objectos, tais como Java, Object Pascal/Delphi, Visual C++, Visual Basic, Corba IDL, COM IDL. É um conceito associado ao desenvolvimento de software baseado em componentes de software. Por exemplo, o Java não tem herança múltipla, o que significa que uma classe apenas estende exactamente uma única (super-)classe. Contudo, uma classe em Java pode imple-

mentar zero, uma ou mais interfaces, pelo que um objecto pode providenciar vários tipos.

Representação Gráfica

Uma interface é representada graficamente em UML como uma classe, mas com estereótipo «interface». Pode ser representada alternativamente na forma compacta pelo seu ícone correspondente (em geral representado por um círculo). A Figura 6.16 ilustra duas representações alternativas para a interface `Enumeration` definida nas bibliotecas standard do Java.

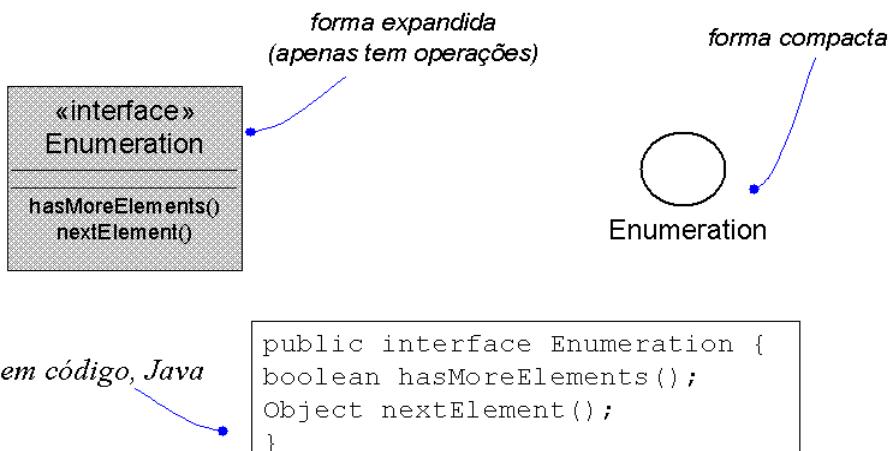



Figura 6.16: Várias representações para a interface `Enumeration`.

Relações entre Interfaces, Classes e Componentes

Tal como acontece com as classes, uma interface pode participar em relações do tipo generalização, associação, e dependência. Adicionalmente pode participar em relações do tipo "realização".

Uma **relação de realização** estabelece-se entre uma interface e uma classe ou entre uma interface e um componente e pode ser apresentada em UML por duas formas alternativas: na forma compacta ou na forma expandida conforme ilustrado na Figura 6.17 relativamente à classe `TargetTracker` e à interface `Observer` definidas num pacote Java do JDK da Sun.

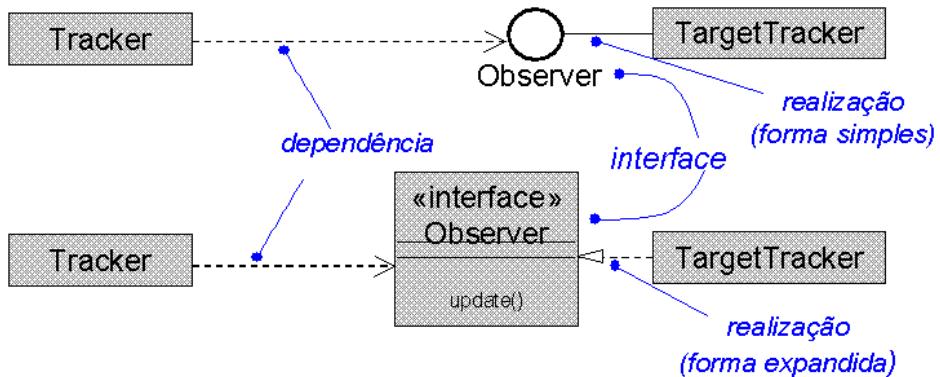



Figura 6.17: Utilização de interfaces em diagramas de classe.

Outro tipo de relação entre classes e interfaces é a relação de dependência, também ilustrada no exemplo da Figura 6.17. A classe `Tracker` depende (ou usa alguma funcionalidade) da classe `TargetTracket` através da interface `Observer`.

A Figura 6.18 ilustra um extracto de um diagrama de componentes, em que apresenta o componente `wordsmith.dll` e todas as suas interfaces exportadas: (1) `IUnknown`, que é uma interface comum a todos os componentes Active-X; (2) `ISpell`, interface com funcionalidades para correcção ortográfica de documentos de texto; e (3) `IThesarus`, interface com funcionalidades de *thesaurus* linguístico (sinónimos, antónimos, etc.). Este exemplo ilustra a importância da noção e utilização de interfaces: em função do acesso ao componente (ou à classe), assim são providenciadas diferentes funcionalidades.

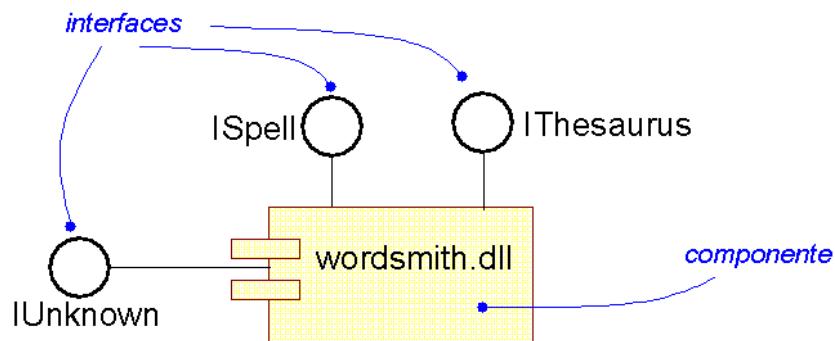



Figura 6.18: Utilização de interfaces em diagramas de componentes.

Outro aspecto fundamental no desenvolvimento de software baseado em componentes, principalmente em contextos distribuídos e empresariais, tem a ver com a gestão de versões de componentes. Por exemplo, a evolução/alteração de um componente pode (e deve) ser realizada de forma transparente, da perspectiva dos restantes componentes/classes (seus clientes), o que é possível caso suporte todas as interfaces definidas em versões anteriores.

6.5 Instâncias e Objectos

Uma **instância** é uma manifestação concreta de uma abstracção, à qual um conjunto de operações pode ser aplicado, e que tem um estado que regista os efeitos das operações realizadas.

Em geral, um "objecto" é designado por "instância" e vice-versa; mas rigorosamente são conceitos distintos. Por exemplo:

- A instância de uma associação é uma ligação (*link*).
- A instância de um nó é um computador que se encontra fisicamente num local.
- A instância de um caso de utilização é um cenário.

Um **objecto** é uma instância de uma classe, herdando, por conseguinte, todos os atributos e métodos definidos na própria classe e possuindo uma representação de execução própria, a qual se pode designar genericamente por estado, bem como uma identificação única no contexto da sua execução. Um objecto em UML é representado, tal como uma classe, através de um rectângulo com uma, duas ou três secções consoante o interesse da informação a apresentar. No entanto, contrariamente às classes, os nomes dos objectos são sublinhados e sufixados pelo nome das classes respectivas, seguindo a notação:

`Nome-do-objecto : Nome-da-classe`

A Figura 6.19 ilustra diferentes exemplos de representação de objectos. Há três objectos com nome (`meuCliente`, `clienteA` e `ClienteB`) e dois objectos anónimos, i.e., sem nome (instâncias das classes `Loja::Virtual` e `AgentNews`). A figura ilustra ainda a representação de instâncias múltiplas (`AgentNews`).

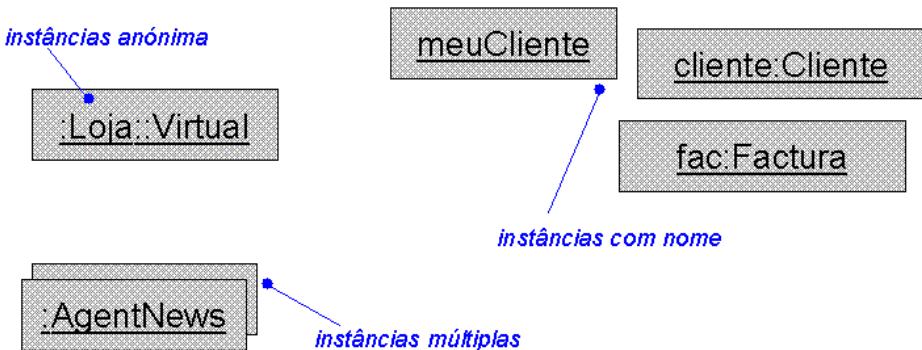



Figura 6.19: Representação gráfica de objectos.

Operações

Os objectos podem efectuar as operações definidas nas suas classes. Por exemplo, considerando a classe `Factura` com a operação `calculaIvaTotal`, se existir o objecto `fac`, então é possível invocar a dita operação do seguinte modo: `fac. CalculaIvaTotal()`.

Por ser redundante não se apresentam as operações de um objecto na sua representação gráfica UML (podem-se contudo apresentá-las em diagramas de interacção – ver Capítulo 7).

Estado

O estado de um objecto é dado pelos valores assumidos pelo conjunto de atributos de um objecto. O estado de um objecto é naturalmente um facto dinâmico, variando ao longo do tempo e do espaço na medida em que o objecto interactua com outros objectos.

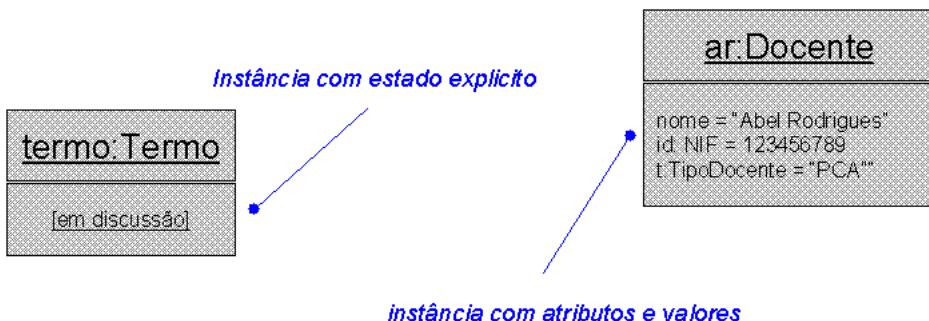



Figura 6.20: Objectos com estado.

A Figura 6.20 ilustra duas formas alternativas de representar o estado de um objecto. O objecto `ar` apresenta o seu estado através da enumeração de valores dos seus respectivos atributos. Note-se que a identificação do tipo dos atributos pode ser realizada (e.g., `id:NIF` ou `t:TipoDocente`) ou não (e.g., `nome`).

Nas situações em que se associa uma máquina de estados a uma determinada classe (e.g., sistemas conduzidos por eventos, ou entidades que evoluem por vários estados ao longo do seu ciclo de vida) pode-se explicitar em determinado momento o estado corrente de um objecto. Na Figura 6.20, o objecto `termo` encontra-se no estado "`em discussão`", que é o nome de um dos estados possíveis definidos na máquina de estados associada à classe `Termo`. Pelo facto de um objecto poder encontrar-se em vários estados simultaneamente, é possível enumerar uma lista de estados correntes.

Objectos Activos

Processos, fluxos de execução, agentes de software e aplicações são exemplos de objectos particulares – designados **objectos activos** – que apresentam uma visão da dinâmica de um sistema. Os objectos activos têm adicionalmente aos objectos (normais ou passivos) a particularidade de terem controlo sobre o seu próprio fluxo de execução.

A especificação do UML sugere que quer as classes quer os objectos activos sejam distinguidos dos restantes pela apresentação de linhas com uma maior espessura (ver parte superior da Figura 6.21).

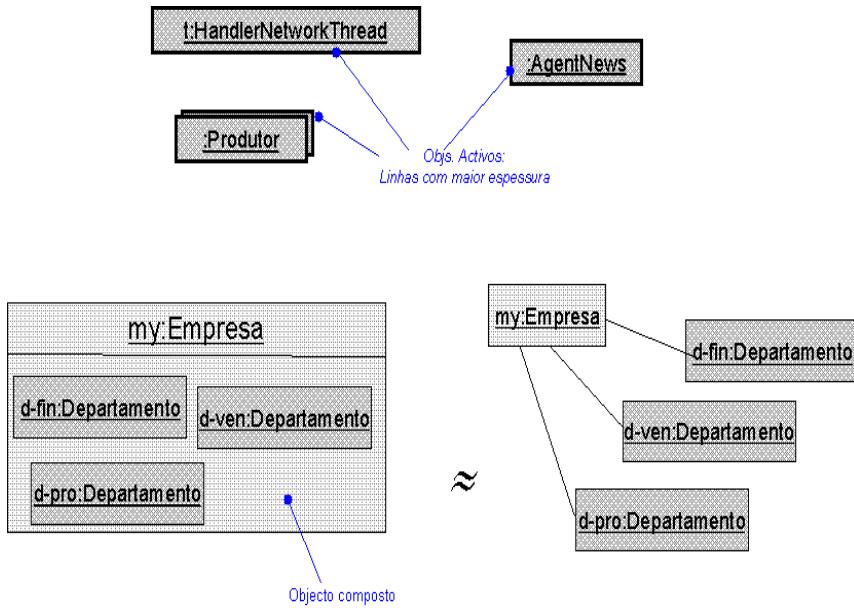



Figura 6.21: Representação gráfica de objectos activos e objectos compostos.

Objectos Compostos

Um **objecto composto** é aquele que é constituído por outros (sub)objectos. Há inúmeras situações de objectos compostos, mas em geral reflectem relações de agregação (simples ou compostas) entre as suas respectivas classes. Os objectos compostos são representados como objectos normais, com a excepção dos seus atributos serem substituídos por outros objectos, usando texto sublinhado ou através de uma representação gráfica.

A parte inferior da Figura 6.21 ilustra um objecto composto relativamente a uma hipotética situação de uma empresa que contem três departamentos. (Consulte-se a Figura 6.10 para se entender as relações de agregação entre os objectos). A vantagem de se representar um conjunto relacionado (por agregação) de objectos através de um objecto composto é essencialmente por motivos de clareza e simplicidade dos diagramas produzidos, para além de se explicitar desta forma o nível de coesão entre os objectos.

6.6 Diagramas de Classes e Diagramas de Objectos

Um **diagrama de classes** ilustra um conjunto de classes, interfaces, colaborações e respectivas relações, em geral de dependência, generalização e de associação. Nas secções anteriores (Secção 6.3 e Secção 6.4) apresentaram-se vários exemplos de diagramas de classes.

Os diagramas de classes são usados para modelar a estrutura de um sistema. Estes modelos são também designados por "vista do desenho estático do sistema" e são usados tipicamente em três situações: (1) para modelar o vocabulário de um sistema; (2) para modelar colaborações simples; e (3) para modelar o desenho de um esquema de uma base de dados.

Um **diagrama de objectos** ilustra um conjunto de objectos e respectivas relações num determinado momento. Permite captar uma imagem ou fotografia momentânea sobre determinado sistema. Um diagrama de objectos é um grafo composto por objectos e ligações (*links*) entre eles. Note-se, como referido acima, que uma ligação é uma instância de uma relação de associação.

Um diagrama de objectos não pode (nem deve pretender) especificar completamente a estrutura de objectos de um dado sistema, já que para cada classe ou conjunto de classes relacionadas, há uma infinidade de potenciais combinações entre as suas instâncias. Por conseguinte, o objectivo dos diagramas de objectos é apenas expor conjuntos relevantes de objectos de modo a melhorar o entendimento das suas funcionalidades e inter-relações.

6.7 Exemplos e Recomendações

De modo a clarificar o que são diagramas de objectos e as suas relações com diagramas de classes, considere-se os seguintes exemplos relativos a exercícios académicos razoavelmente simplificados: (6.1) sistema de gestão de automóveis; (6.2) sistema de gestão de compras; e (6.3) sistema académico.

Exemplo 6.1: Diagrama de Classes/Objectos do Sistema de Gestão de Automóveis

Considere-se o seguinte enunciado: "*Uma pessoa pode ser proprietário de vários veículos e estes são possuídos apenas por uma única pessoa. Por outro lado, um veículo tem de possuir necessariamente um motor. Um veículo é identificado univocamente pela matrícula e possui ainda outras informações, tais como a cor, data de fabrico, marca e modelo. Um motor é identificado por um número de motor, tipo de combustível e cilindrada*".

A Figura 6.22 ilustra o diagrama de classes correspondente ao enunciado introduzido. Note-se que poderiam ser produzidas algumas variantes a esta solução. Por exemplo, poder-se-ia ter considerado a marca, modelo e tipo de combustível como classes no sistema.

Note-se que a relação entre motor e veículo é de agregação (pois um motor é visto como um componente de um veículo), mas não deve ser composição, ou agregação composta, pois podem existir motores sem estarem directamente colocados nos veículos (estão algures em stock à espera, por exemplo, de serem adquiridos e substituírem um motor gripado!).

Figura 6.22: Diagrama de classes do Exemplo 6.1.

Com base no enunciado e no diagrama de classes acima a Figura 6.23 apresenta o diagrama de objectos que traduz a seguinte situação: *"o Zé Maria é dono de um Audi A3 TDi vermelho, com matricula '99-99-MM', que tem um motor 1900cc, com número '9999'".* Note-se que no diagrama de objectos apenas são representados objectos e ligações, e nestas últimas não se apresentam quaisquer adornos do tipo multiplicidade, agregação ou visibilidade. Pode-se quanto muito atribuir um nome à ligação para facilitar a sua legibilidade.

ZéMaria:Pessoa
nome="Zé Maria"
tem
AudiA3:Veiculo
modelo=A3 TDi
matricula=99-99-MM
cor=vermelho
tem
:Motor
número=9999
cilindrada=1900cc
combustível=gasóleo

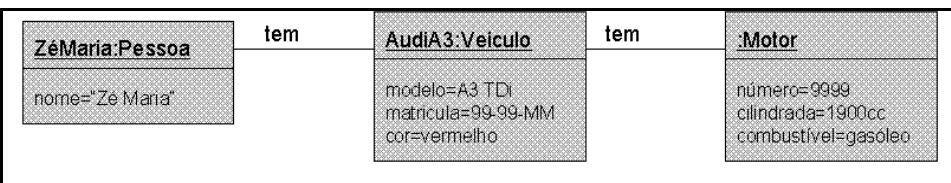



Figura 6.23: Diagrama de objectos do Exemplo 6.1.

Fica como exercício e reflexão a construção do diagrama de objectos que traduza a seguinte situação: "*o Zé Maria e a Rita são donos de um Audi A3 Tdi vermelho, com matricula '99-99-MM', que tem um motor 1900cc, com número '9999'*". Que alterações ao diagrama de classes teriam de ser realizadas de modo a validar esta nova situação?

Exemplo 6.2: Diagrama de Classes/Objectos do Sistema de Gestão de Compras.

Considere-se o seguinte enunciado: "*A empresa XPTO compra produtos a diferentes fornecedores. Os produtos adquiridos são identificados univocamente por um código, têm uma descrição, e ainda a identificação de um tipo de produto (e.g., alimentar, vestuário, linha branca). Cada encomenda especifica um conjunto de produtos com respectivas quantidades, o fornecedor, a data de aquisição, e a data de entrega prevista…*".

A Figura 6.24 ilustra o diagrama de classes de alto nível correspondente ao enunciado introduzido.

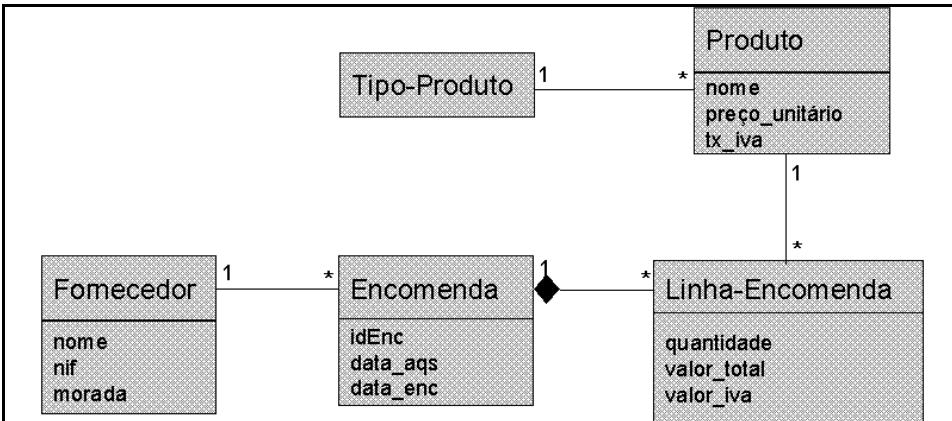



Figura 6.24: Diagrama de classes do Exemplo 6.2.

Facto Real: A Figura 6.25 (é uma variante da Figura 6.24) ilustra dois erros usuais na construção deste tipo de diagrama de classes de alto nível:

- Introdução de classes específicas de estruturas de dados do tipo contentores (e.g., listas, *hashtables*, conjuntos) ao nível da análise do modelo de dados.
- Especificar atributos de chaves estrangeiras entre classes. Esses atributos existem, mas de forma implícita nas associações envolvidas.

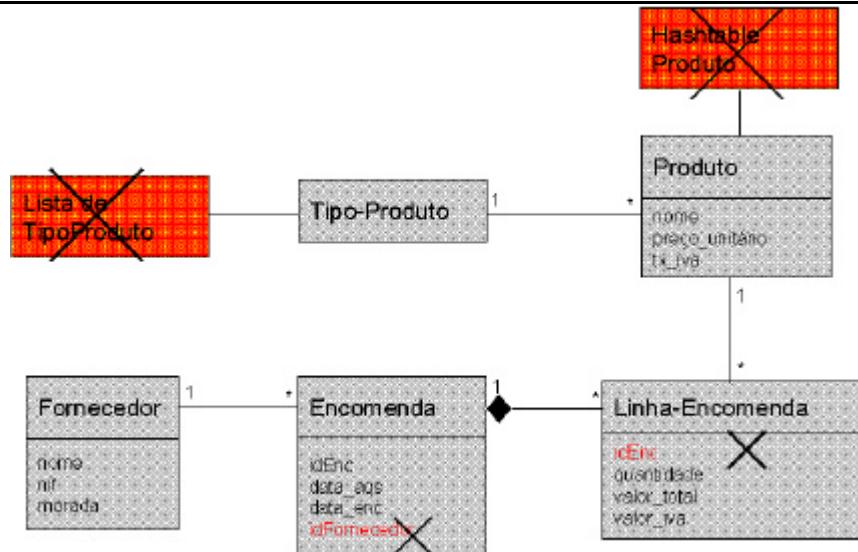



Figura 6.25: Diagrama de classes do Exemplo 6.2, com erros usuais.

Com base no enunciado e no diagrama de classes da Figura 6.24 apresenta-se na Figura 6.26 o diagrama de objectos que traduz a seguinte situação "*A Nestlé satisfez a encomenda 333, em 99/10/14, relativa à data de encomenda de 99/8/31. A encomenda 333 tem 2 itens: (i) produto 123, chocolate BLO, Euro 30; 10000 unidades; e (ii) produto 135, leite condensado 1/4, Euro 20, 50000 unidades. Ambos os produtos são do tipo alimentar*".

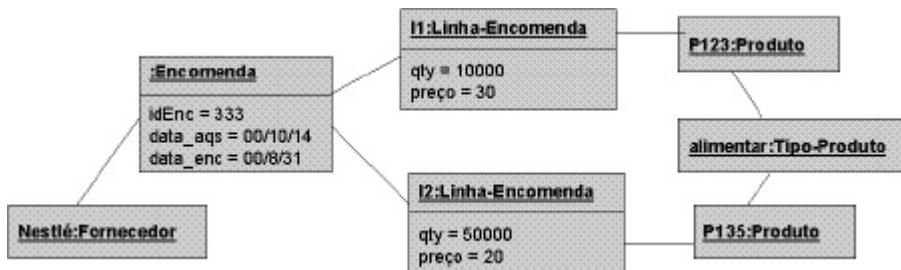



Figura 6.26: Diagrama de objectos do Exemplo 6.2.

Note-se uma vez mais que um diagrama de objectos não é mais que uma possível configuração ou malha de objectos; é portanto, apenas um de uma infinidade de possíveis disposições.

Exemplo 6.3: Diagrama de Classes/Objectos do Sistema Académico.

Considere o seguinte diagrama de classes, ilustrado na Figura 6.27, correspondente aos seguintes factos: (1) "*um docente dar aulas a vários alunos numa ou mais salas*" e (2) "*um docente, enquanto professor, ser responsável por outros docentes, designados por assistentes*".

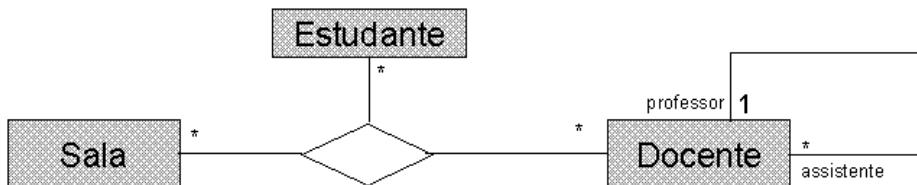



Figura 6.27: Diagrama de classes do Exemplo 6.3.

A Figura 6.28 ilustra dois diagramas de objectos desenvolvidos sobre o diagrama de classes anterior, e relativamente aos dois cenários seguintes:

1. *"O docente 'Alberto' dá aulas na sala 'P01' a vários alunos"*. Note-se em particular a forma como se representa "*vários alunos*" através da especificação de instâncias múltiplas e anónimas da classe `Estudante`. Outro aspecto relevante é a forma como se representa uma configuração de objectos cujas classes respectivas se encontram numa relação N-ária.
2. "*O docente 'Alberto' é responsável pelo docente (enquanto assistente) 'Pedro'*".

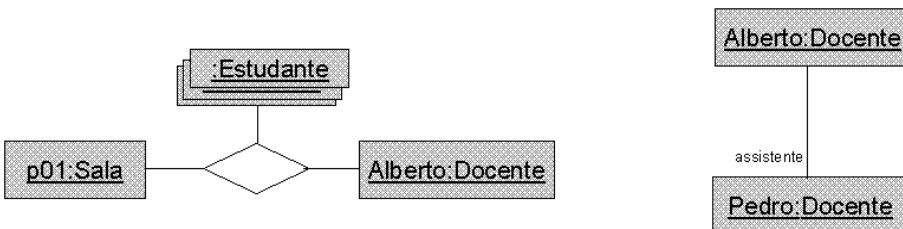



Figura 6.28: Diagrama de objectos do Exemplo 6.3.

6.8 Exercícios

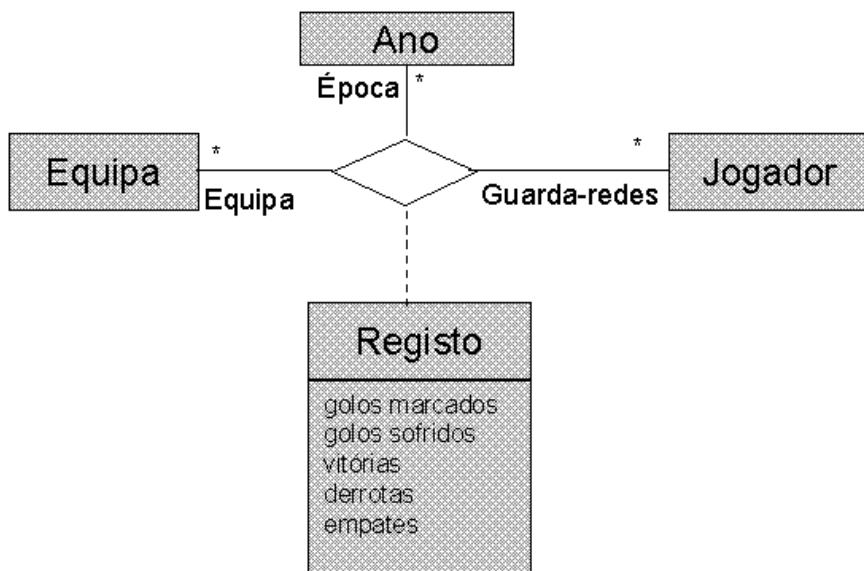
Ex.38. Usar classes para definir o glossário do sistema “Jogo de Futebol” descrito de seguida: “O jogo de futebol é realizado por duas equipas de jogadores. Cada equipa é composta por 11 jogadores, com diferentes funções: o guarda-redes, defesas, médios, atacantes, e pontas de lança. O ponta de lança é um atacante especial por ter especiais características de goleador... O jogo é realizado num campo com medidas regulamentares (em comprimento e largura), tem duas balizas, cada qual em extremos opostos do campo. Ganha o jogo a equipa que marcar mais golos (i.e., colocar a bola) na baliza do adversário. No jogo apenas existe uma única bola, que apresenta características (peso, diâmetro, …) regulamentares... O jogo de futebol é mediado por uma equipa de 3 árbitros, em que um é o árbitro principal, e os outros dois são árbitros auxiliares…”.

Ex.39. Tendo em conta o sistema “Jogo de Futebol” descrito no exercício anterior e as classes identificadas estabeleça agora as suas relações de forma a descrever o modelo de classes correspondente.

Ex.40. Considere o diagrama de classes relativo ao sistema de “Jogo de Futebol” produzido no exercício anterior. Defina 4 pacotes respectivamente para agrupar classes relativas a (1) equipas de jogadores; (2) equipas de arbitragem; (3) clubes de futebol; e (4) os jogos propriamente ditos. Defina o diagrama de pacotes respectivo, evidenciando as classes exportadas e as dependências de importação correspondentes.

Ex.41. Tendo em conta o Exemplo 6.1, defina o diagrama de classes e o diagrama de objectos que suportem as seguintes afirmações:

(1) “*o empresa XPTO possui um Audi A6 TDi vermelho, com matricula ‘99-99-AA’, que tem um motor 1900cc, com número ‘9999’*”.

(2) “*a Marta é dona de um Ferrari F40 vermelho, com matricula ‘66-66-FF’, mas sem motor*”.

(3) "*o Rui não têm qualquer carro*".

Ex.42. Modelize através de um diagrama de classes o seguinte discurso: "*Uma mesa de café é constituída por um tampo e por quatro pernas…*".

Ex.43. Considere o seguinte discurso relativamente a um sistema de partidas de ténis: "Num torneio de ténis, cada partida é jogada entre 2 jogadores. Pretende-se manter informação sobre o nome e idade dos jogadores; data da partida e atribuição dos jogadores às partidas. O máximo de partidas que um jogador poderá realizar é 6 e o mínimo 1". Pretende-se:

(1) O diagrama de classes correspondente.

(2) O diagrama de objectos que ilustre a seguinte situação: *"Os jogadores Zé Maria e Pedro Cunha disputaram uma partida às 20:30 de 2001/10/10"*.

Ex.44. Observe atentamente o seguinte diagrama de classes e indique textualmente o seu significado.

Ex.45. Modelize através de um diagrama de classes UML os conceitos relativos a um sistema básico de "Gestão de Facturas": "Um sistema de facturação mantém informação sobre clientes, sobre tipos de produtos e de serviços vendidos/prestados. Um cliente é identificado univocamente pelo NIF, e tem ainda nome, morada, telefones, e tipo de cliente. Um cliente pode ter mais que uma morada… Uma factura é identificada univocamente pelo IDFactura, que é um número sequencial. Tem ainda a informação sobre data de facturação, cliente, valor total. Uma factura tem várias linhas, cada qual especificando a venda de um bem ou serviço… Uma factura pode ser paga por uma ou mais prestações. Cada pagamento parcial/total corresponde à emissão de respectivo recibo...".

Ex.46. Considerando o modelo de classes resultante do exercício anterior ("Gestão de Facturas") descreva através de diagramas de objectos as seguintes situações:

(1) "*O cliente IPP S.A., com NIF '123456789', com duas moradas. A primeira na 'Praça da Alegria, 33, 1300-222 Lisboa' e a segunda na 'Rua da Paz, 44, 4ºEsq, 2000-320 Santarém*".

(2) "*A factura, n.º "3445/2000", data de facturação em "28/11/2000", cliente "IPP S.A., e valor total de "350,000$00, é constituída por duas linhas. A primeira linha de factura consiste na venda de "200 caixas de parafusos de 20'"; a segunda linha consiste na venda de '10 perfuradoras de 350W'"*

Ex.47. Considere a seguinte extracto de código Java relativo utilização de classes definidas no pacote `java.sql.*`, em particular das classes `DriverManager`, `Connection` e `Statement`. Considere ainda que o código ilustrado está implementado na classe `Cliente`. Desenhe o diagrama de classes de suporte e o diagrama de objectos correspondente.

```
            Connection con;
            Statement stmt;
            ...
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            con                                              =
DriverManager.getConnection("jdbc:odbc:BD1");
            stmt = con.createStatement();
            ...
            stmt.executeUpdate("INSERT …");
            ...
            stmt.executeUpdate("UPDATE …");
```

Capítulo 7 - UML – Modelação do Comportamento

Tópicos

- Introdução
- Interacções
- Diagramas de Interacção
- Diagramas de Estados
- Diagramas de Actividades
- Exercícios

7.1 Introdução

Em qualquer sistema minimamente interessante, os objectos não estão estáticos, mas interagem entre si por troca de mensagens [Booch99].

A modelação do comportamento de um sistema de software consiste, segundo a abordagem orientada por objectos, em dois tipos distintos de especificações. Por um lado, na **modelação do comportamento inter-objectos**, ou seja na identificação dos seus padrões de trocas de mensagens (diagramas de interacção). Por outro lado, na **modelação do comportamento intra-objecto**, ou seja na identificação dos estados em que um objecto se pode encontrar ao longo do seu ciclo de vida, dos eventos envolvidos, bem como dos seus algoritmos de implementação (diagramas de estados e de actividades).

A modelação de um sistema de software com base em diagramas de classes e de objectos traduz apenas as suas relações estruturais e estáticas. Nada é revelado sobre o padrão interno e externo do comportamento dos objectos ou relativamente à definição de determinado algoritmo. O UML providencia os seguintes elementos que permitem a especificação do comportamento de um sistema de software: objectos, interacções, mensagens, estados, actividades, sinais e eventos. Com base nestes elementos podem-se definir os seguintes tipos de diagramas: diagramas de interacções, diagramas de estados e diagramas de actividades.

7.2 Interacções

Uma **interacção** é a especificação do comportamento de um conjunto de instâncias, representado pela sua troca de mensagens, num determinado contexto, e com vista à concretização de um dado objectivo.

Embora uma interacção permita descrever o comportamento de instâncias genericamente (e.g., objectos, cenários de casos de utilização, instâncias de actores), passaremos a utilizar de ora em diante a designação "objecto" de forma a simplificar a explanação.

As interacções são utilizadas para modelar o fluxo de controlo de uma operação, classe, componente, caso de utilização ou um sistema na sua globalidade. Sempre que existe uma ligação (link) entre instâncias, pode ocorrer uma ou mais interacções.

Uma mensagem define uma comunicação particular entre instâncias, que é especificada numa determinada interacção. Uma comunicação pode ser, por exemplo: invocar uma operação; lançar um sinal; construir ou destruir uma instância. Uma mensagem especifica não apenas o tipo de comunicação, mas também os papéis do emissor e receptor, a acção lançada, bem como o papel desempenhado pela ligação.

7.2.1 *Objectos e Ligações*

Pode-se encarar um diagrama de objectos como a representação dos aspectos estáticos de uma interacção. Contudo, uma interacção vai mais longe, ao introduzir uma sequência dinâmica de mensagens que podem fluir entre esses objectos. Desta forma, os diagramas de interacção devem ser considerados como uma extensão aos diagramas de objectos.

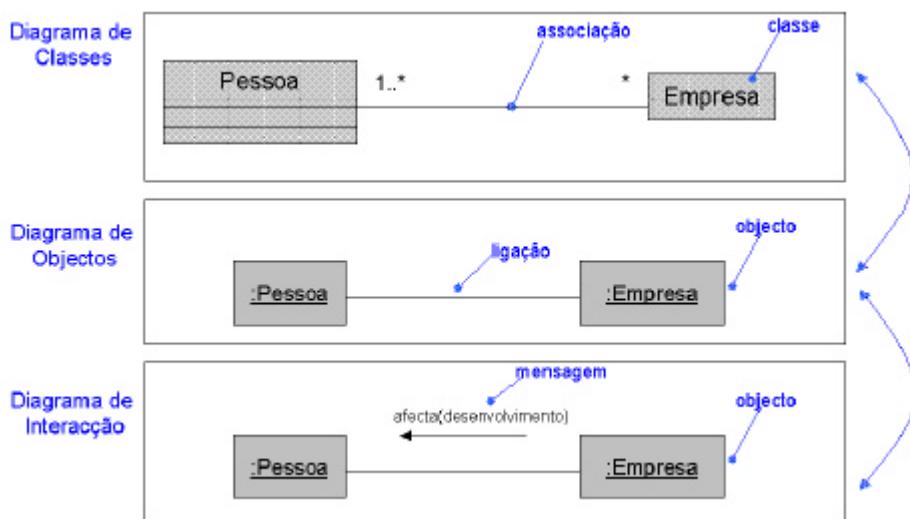



Figura 7.1: Relações entre diagramas de classes, de objectos e de interacções.

A Figura 7.1 ilustra a relação entre os conceitos mensagem, ligação e associação, tal como as relações entre diagramas de classes, de objectos e de interacção.

Uma **ligação** (*link*) entre objectos traduz a existência de uma relação semântica ou estrutural entre as suas respectivas classes. Sempre que existe uma associação entre duas classes deve também existir uma ligação entre instâncias das classes respectivas. Uma ligação entre objectos é representada em UML através de uma linha a cheio, não dirigida.

(Nota: Na ferramenta Rational Rose não existe possibilidade de representar explicitamente diagramas de objectos. Tal situação é perfeitamente admissível caso se atente na afirmação realizada anteriormente: os diagramas de objectos são um caso simplificado dos diagramas de interacções.)

Facto Real: Num projecto de um curso de pós-graduação pedi aos alunos para especificarem um sistema segundo diferentes perspectivas. Entre outros, pedi para que fizessem um diagrama de objectos. Resposta de alguns alunos: *"Não fizemos pois a ferramenta (Rational Rose) não permite realizar tal tipo de diagrama…"*. Ou seja, os alunos deviam ter produzido o diagrama de objectos pedido, usando o diagrama de interacção (em particular um diagrama de colaboração), explicitando apenas objectos e suas ligações.

7.2.2 Mensagens e Estímulos

Um **estímulo** é uma comunicação entre dois objectos que veicula informação com a expectativa que determinada actividade seja realizada. Um estímulo provocará a invocação de uma operação, o envio de um sinal, ou a criação ou destruição de um objecto.

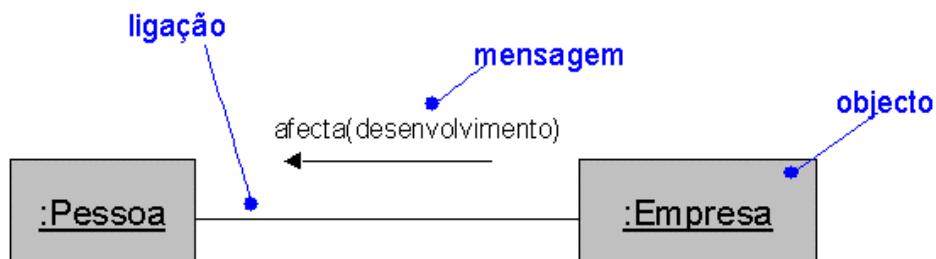



Figura 7.2: Diagrama de interacções.

Uma **mensagem** é a especificação de um estímulo, ou seja, específica os papéis que os objectos emissor e receptor devem acordar para que uma acção seja realizada. Regra geral, a recepção de uma mensagem resulta na realização de uma acção (que por sua vez pode provocar uma mudança de estado) no objecto destino.

7.2.3 Representação de Mensagens

Em geral um estímulo é representado por uma linha sólida dirigida (i.e., uma seta) do objecto origem para o objecto destino. Pode-se, contudo, usar variantes a esta representação principal para ilustrar diferentes tipos de comunicações (ver Figura 7.3):

- Simples
  Fluxo de controlo simples. Cada seta mostra o próximo passo numa determinada sequência. Quando não é relevante identificar o tipo de comunicação deve-se usar este tipo de seta.
- Síncrona
  Invocação de método ou outro fluxo de controlo, dentro do fluxo corrente. Pode ser usado em situações normais de invocação de métodos. Pode também ser usado entre objectos activos concorrentes, quando um invoca um sinal e espera que o comportamento destino termine.
- Assíncrona
  Usado alternativamente à seta do tipo simples para ilustrar explicitamente uma comunicação assíncrona entre dois objectos numa sequência procedimental.
- Retorno
  Retorno de uma invocação de método.

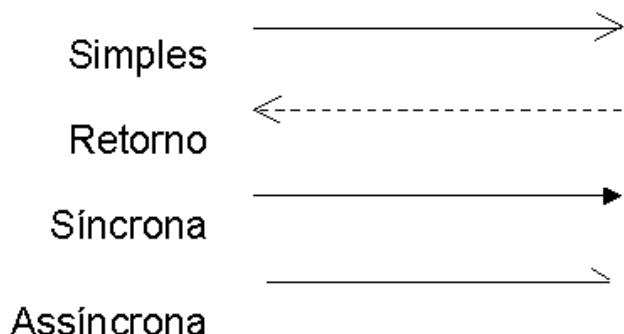



Figura 7.3: Diferentes representações de mensagens.

Num fluxo de controlo procedimental, a seta de retorno pode (e deve) ser omitida, já que corresponde implicitamente ao fim da respectiva activação. O valor de retorno, caso exista pode ser ilustrado na seta inicial. Por outro lado, num fluxo de controlo não-procedimental (e.g., em mensagens assíncronas e processamento paralelo), o retorno deve ser ilustrado explicitamente.

Num sistema concorrente, a seta simples representa "passar" o fluxo de controlo para a outra actividade (semântica bloqueante) enquanto a seta assíncrona representa o envio de uma mensagem sem "passar" o seu fluxo de controlo (semântica não bloqueante).

7.2.4 Tipos de Mensagens

Em UML estão predefinidos diferentes tipos de mensagens:

- *Call*: Invoca uma operação de um objecto. É o tipo de mensagem mais comum (comunicação síncrona).
- *Return*: Devolve um resultado/sinal para o objecto "chamador".
- *Send*: Envia um sinal para um objecto (comunicação assíncrona).
- *Create*: Cria um objecto.
- *Destroy*: Destroi um objecto. Note-se que um objecto pode destruir-se a si próprio.

A Figura 7.4 ilustra um exemplo de aplicação destes diferentes tipos de mensagens. Note-se que apenas se representam explicitamente os estereótipos das mensagens do tipo criação e destruição de objectos. Todas os outros tipos (*call*, *send* e *return*) são implícitos a partir da sua representação gráfica. Note-se ainda na representação gráfica particular para a destruição de um objecto ("X").

7.3 Diagramas de Interacção

Quando um objecto envia uma mensagem a outro objecto, este por sua vez pode enviar uma outra mensagem a outro objecto, e assim sucessivamente. Criam-se deste modo sequências de mensagens executadas, em geral, sobre o mesmo processo ou actividade de execução.

Uma interacção é representada por um **diagrama de interacção**. Os diagramas de interacção são usados para especificar a realização de um caso de utilização bem como a realização de uma operação envolvendo diferentes objectos. Os diagramas de interacção podem ser apresentados nas seguintes formas:

- Diagrama de sequência: é um diagrama de interacção com ênfase na ordenação temporal das mensagens trocadas entre os objectos (ver Figura 7.4).
- Diagrama de colaboração: é um diagrama de interacção com ênfase na organização estrutural dos objectos que trocam mensagens entre si (ver Figura 7.5).

Os diagramas de sequência são particularmente úteis para detalhar um cenário de um caso de utilização, e são mais adequados para especificar situações complexas, bem como múltiplos e concorrentes fluxos de controlo. Por outro lado, os diagramas de colaboração ao colocarem a ênfase nas relações estruturais entre as instâncias de uma interacção, são mais adequados no desenho de sistemas não concorrentes (i.e., desenho de interacções sequenciais ou procedimentais) e em particular para ilustrar relações entre objectos em padrões de desenho [Gamma94].

Uma colecção de restrições standard pode ser usada para ilustrar o momento de criação ou destruição de objectos ou ligações durante uma determinada execução:

- Objectos e ligações criados durante uma execução podem ser designados por {*new*}.
- Objectos e ligações destruídos durante uma execução podem ser designados por {*destroy*}.
- Objectos e ligações criados durante uma execução e seguidamente destruídos podem ser designados por {*transient*}.

Podem-se introduzir nos diagramas de interacção várias descrições (e.g., restrições temporais, descrições de acções durante determinada activação), as quais devem ser colocadas na margem (esquerda ou direita) ou junto às activações ou transições que representam.

7.3.1 Diagramas de Sequência

Um **diagrama de sequência** ilustra uma interacção segundo uma visão temporal. Um diagrama de sequência é representado através de duas dimensões: a dimensão horizontal, que representa o conjunto de objectos intervenientes; e a dimensão vertical que representa o tempo. A apresentação destas dimensões pode ser invertida, se for conveniente. Não existe qualquer significado na ordenação horizontal dos objectos intervenientes, ou seja, na sua disposição relativa.

Nos diagramas de sequência as setas são desenhadas horizontalmente de forma a representar a indivisibilidade da operação necessária para enviar o estímulo. Esta assunção é válida para a generalidade das situações. Todavia, em situações em que o tempo de envio do estímulo não é negligenciável (e.g., pode ocorrer um outro estímulo em sentido oposto), a seta deve ser representada de modo relativamente inclinada para baixo (i.e., no sentido do tempo).

A Figura 7.4 ilustra um diagrama de sequência em que são representados diferentes tipos de mensagens. Note-se em particular a criação e destruição do objecto `:Assistente`; a explicitação de focos de controlo (ou zonas de activação), e o envio de mensagem assíncrona `notifica` do objecto `:Cliente` para o objecto `:AgênciaViagem`.

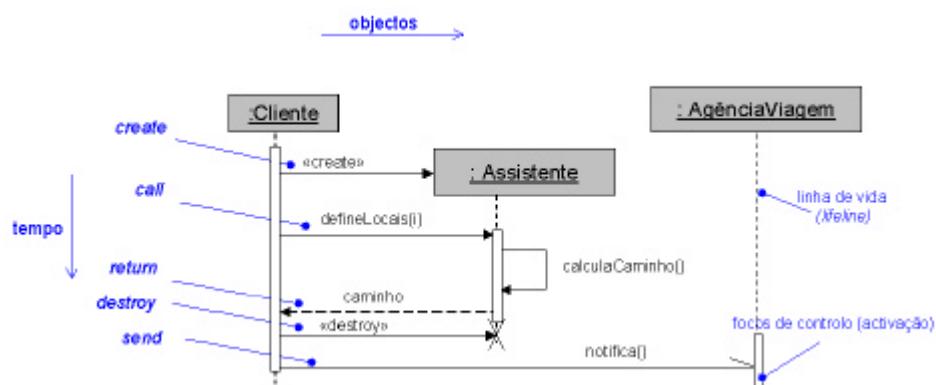



Figura 7.4: Exemplo de um diagrama de sequência.

7.3.2 Diagramas de Colaboração

Um **diagrama de colaboração** ilustra uma interacção organizada espacialmente. De forma distinta dos diagramas de sequência, um diagrama de colaboração mostra as relações entre objectos que desempenham diferentes papéis. Por outro lado, um diagrama de colaboração não mostra o tempo como uma dimensão separada, pelo que a sequência de interacções e de actividades concorrentes é representada usando-se números sequenciais.

A ordem de uma interacção é descrita através de uma sequência de números, normalmente com início em 1. Num fluxo de controlo procedimental, os números de comunicação de uma subsequência são representados de acordo com o respectivo nível de inclusão. Para uma sequência de interacções não procedimental, i.e., entre objectos concorrentes, todos os números de uma sequência encontram-se ao mesmo nível.

A Figura 7.5 ilustra um exemplo de diagrama de colaboração na forma de diagrama de instâncias.

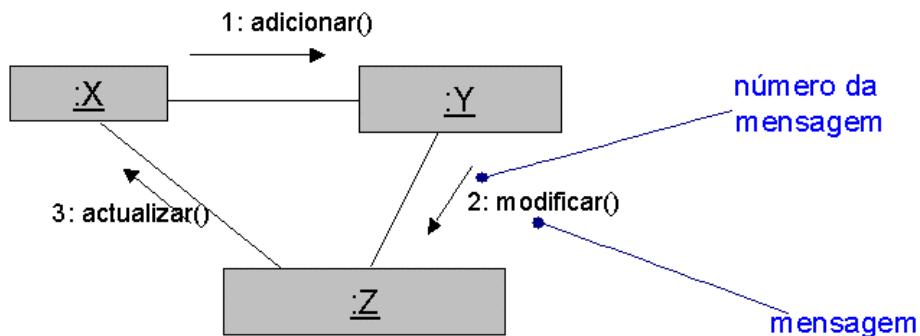



Figura 7.5: Exemplo de um diagrama de colaboração.

Um diagrama de colaboração pode ser representado por duas formas: ao **nível de especificação** (o diagrama ilustra os papéis que as classes e associações desempenham, bem como as suas mensagens) ou ao **nível de instância** (o diagrama ilustra objectos, ligações e estímulos). A primeira forma apresenta os papéis e estrutura definida na colaboração

subjacente, enquanto que a segunda ilustra uma instância que deve ser conforme com os papéis de uma colaboração.

Exemplo 7.1: Diagramas de Colaboração – Pessoa com distintos Papéis.

Considere-se o seguinte enunciado: "*Num contexto académico, uma pessoa pode desempenhar dois papéis distintos. Por um lado, uma pessoa, como professor, pode ser o regente ou coordenador de (zero ou mais) disciplinas e pode ser responsável pela supervisão de (zero ou mais) estudantes. Por outro lado, uma pessoa como estudante tem necessariamente um tutor (o professor que o supervisiona), e inscreve-se em (zero ou mais) disciplinas*". Mostra-se neste exemplo as relações entre diagramas de classes, de colaboração de nível específico, e de colaboração de nível de instâncias.

A Figura 7.6 ilustra o diagrama de classes correspondente ao enunciado introduzido. Note-se a associação reflexiva `supervisionar` que relaciona pessoa, no papel de professor, com pessoa, no papel de estudante. Note-se também nas duas associações (`inscrever` e `coordenar`) entre as classes `Pessoa` e `Cadeira`.

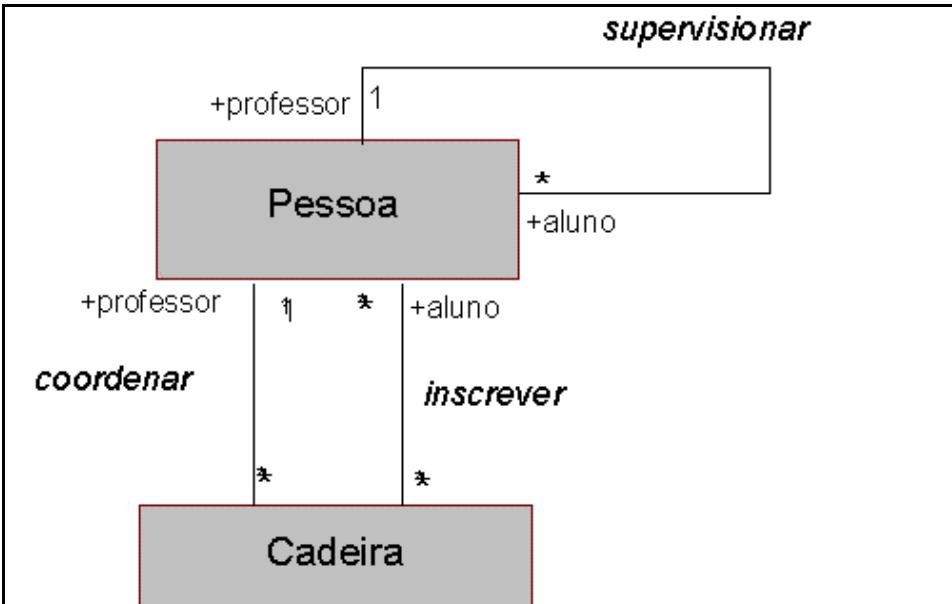



Figura 7.6: Diagrama de classes do Exemplo 7.1.

A Figura 7.7 ilustra o diagrama de colaboração de nível de especificação. Note-se que são representados os papéis que as classes e associações desempenham genericamente numa colaboração. Note-se que cada rectângulo apresenta o nome de uma classe precedida pelo caracter "**:**" e eventualmente pelo nome do papel que a classe desempenha nessa colaboração precedida pelo caracter "/". (Para mais detalhes consulte-se [OMG99].)

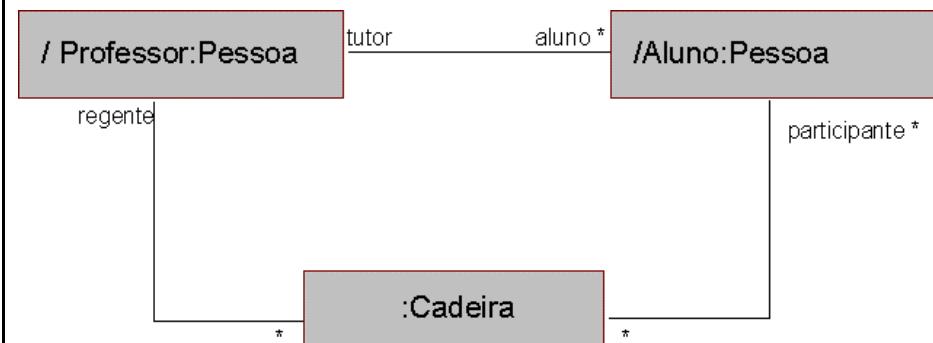
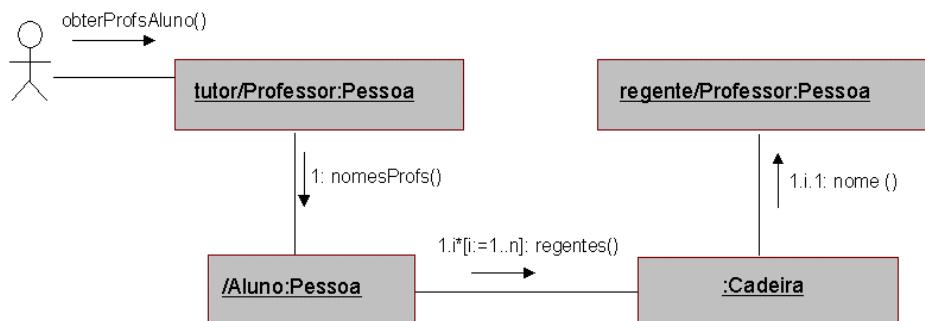



Figura 7.7: Diagrama de colaboração de nível de especificação do Exemplo 7.1.

Por fim, a Figura 7.8 ilustra um diagrama de colaboração, de nível de instâncias, conforme com o anterior diagrama, correspondente à operação de "obtenção de todos os professores de um dado aluno" (i.e., professores regentes em cadeiras inscritas pelo respectivo aluno) realizada sobre instâncias de professor enquanto tutor.

Figura 7.8: Diagrama de colaboração de nível de instâncias do Exemplo 7.1.

Note-se na representação da sequência de execução de operações: fluxo de controlo sequencial com uma iteração (envio da mensagem `regente` para todas as instâncias de `Cadeira`, correspondentes às cadeiras em que os alunos se encontram inscritos).

7.3.3 Equivalência Semântica

A equivalência semântica entre os diagramas de sequência e de colaboração significa que estes representam (ou podem representar) a mesma interacção. O Exemplo 7.2 ilustra a equivalência semântica entre estes tipos de diagramas. A ferramenta Rational Rose, por exemplo, gera automaticamente um diagrama a partir do outro e vice-versa.

Exemplo 7.2: Equivalência entre Diagramas de Interacção – Acesso a BD em Java.

Considere o seguinte extracto de código Java relativo à utilização de classes definidas no pacote `java.sql.*`, em particular das classes `Connection` e `Statement`. Pretende-se os respectivos diagramas de sequência e de colaboração.

```
Connection con;   Statement stmt;
...
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
con =
DriverManager.getConnection("jdbc:odbc:BD1");
stmt = con.createStatement();
...
stmt.executeUpdate("INSERT …");
stmt.executeUpdate("UPDATE …");
```

Para resolver este exemplo, é necessário antes de mais definir o diagrama de classes respectivo (ver Figura 7.9).

Existem três classes principais: a classe `DriverManager` que é responsável por gerir os detalhes de acesso a determinado sistema de base de dados através de um determinado controlador e adicionalmente permite criar conexões (através do método estático `getConnection`); a classe `Connection`, responsável por estabelecer uma ligação a determinada base de dados; e a classe `Statement` que encapsula os detalhes de uma instrução SQL. Sobre um `DriverManager` podem ser criadas instâncias de `Connection`, e sobre esta diferentes instâncias de `Statement`.

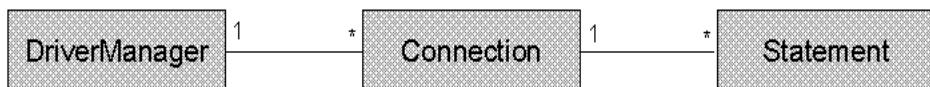



Figura 7.9: Diagrama de classes do Exemplo 7.2.

Com base no diagrama de classes, que ilustra as relações estruturais entre as diferentes classes pode-se definir os diagramas de interacção correspondentes ao código ilustrado.

Note-se que neste exemplo temos explicitamente os objectos `con` e `stmt`, (respectivamente instâncias de `Connection` e de `Statement`); e que não existe uma instância explícita de `DriverManager`: já que é a própria classe que providencia um método-fábrica de ligações.

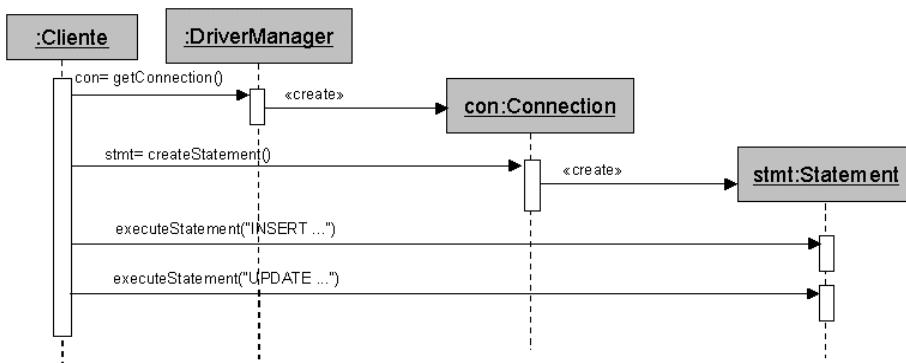



Figura 7.10: Diagrama de sequência do Exemplo 7.2.

Atente-se na equivalência semântica entre ambos os diagramas (ver Figuras 7.10 e 7.11). É introduzido genericamente uma instância da classe `Cliente` de modo a ilustrar devidamente o padrão de interacção.

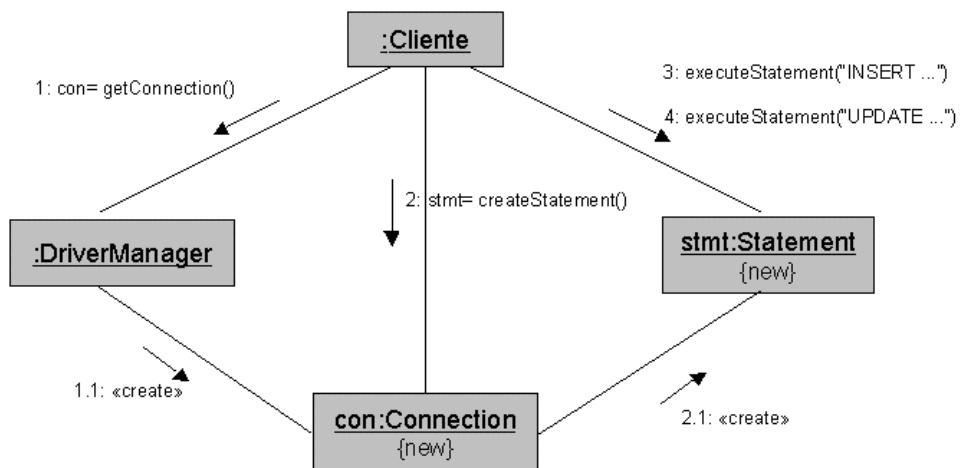



Figura 7.11: Diagrama de colaboração do Exemplo 7.2.

7.3.4 Diagramas de Interacção e de Casos de Utilização

Viu-se no Exemplo 7.2 uma possível utilização dos diagramas de interacção: para modelar (ao nível de desenho) uma determinada interacção, ou conjunto de interacções, entre várias instâncias.

Outra das aplicações típicas destes diagramas é a especificação (ao nível de análise) de um caso de utilização. Ou seja, complementarmente à especificação do caso de utilização em linguagem natural (em texto, mais ou menos estruturado), o mesmo poderá, em determinadas situações, ser clarificado através de um ou mais diagramas de interacção.

Exemplo 7.3: Diagramas de Interacção para descrever Casos de Utilização.

Considere o exemplo do sistema da "Máquina de Bebidas" descrito na Secção 5.4. Considere para simplificar o caso de utilização "Comprar Bebida". Pretende-se neste exemplo especificar o cenário ideal (em que tudo corre bem, i.e., em que há bebida, há troco, etc.) deste caso através de diagramas de interacção.

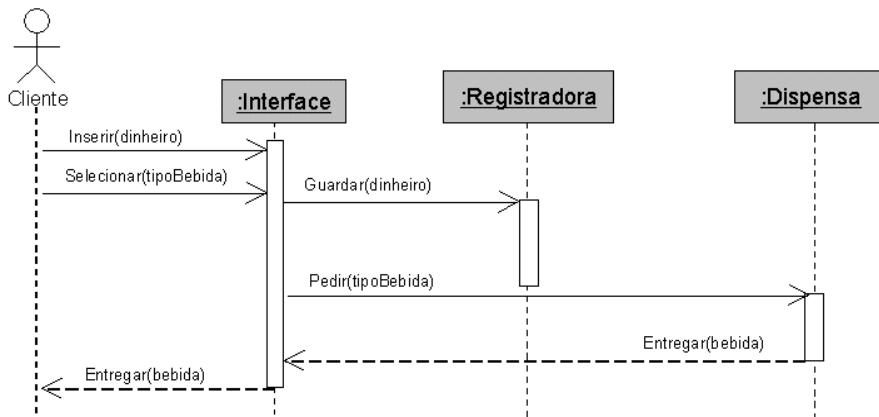
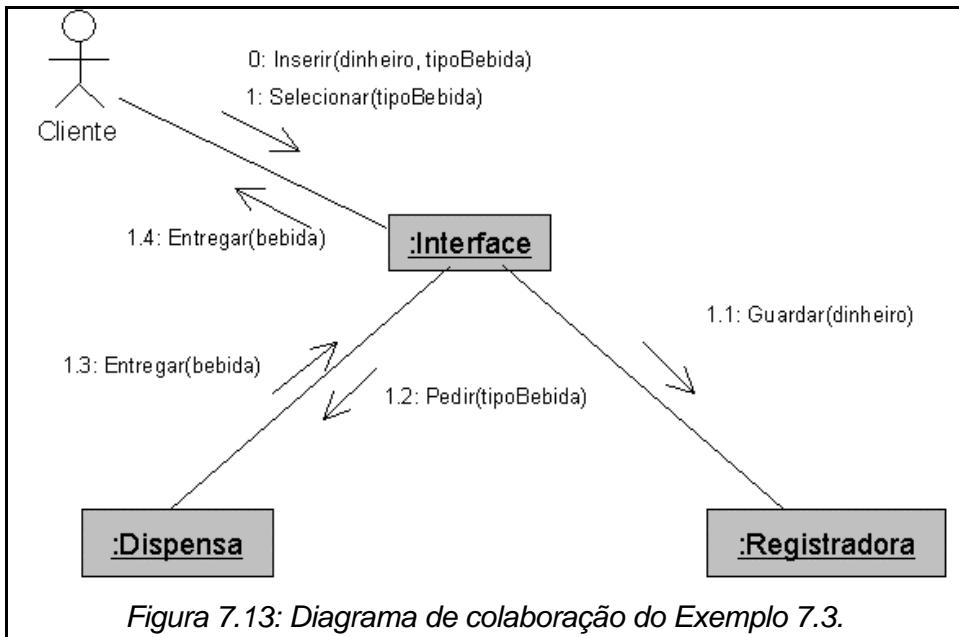



Figura 7.12: Diagrama de sequência do Exemplo 7.3.

Neste tipo de problema, é necessário identificarem-se os objectos que deverão de alguma forma interagir. Considere para o efeito que a máquina é composta, entre outros, por três objectos principais:

- Interface: o painel de interface com o utilizador
- Registadora: a caixa registadora, que guarda o dinheiro
- Dispensa: a caixa/armário que guarda as diferentes bebidas

Considere ainda que o cenário a representar é composto pela seguinte sequência de acções:

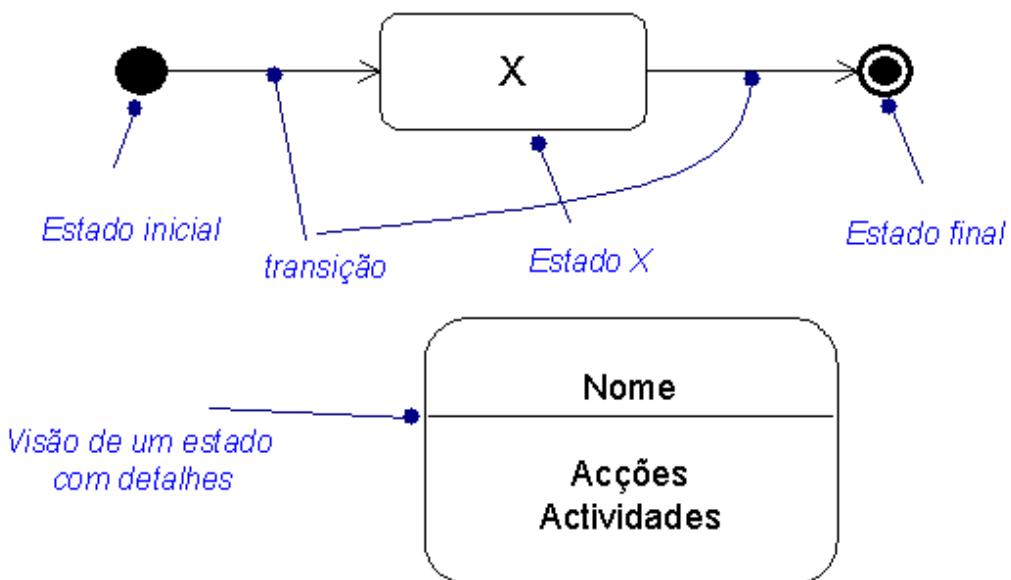
- O cliente insere o dinheiro na ranhura no painel de interface da máquina
- O cliente selecciona o tipo de bebida
- O dinheiro "vai até" a caixa registradora, esta actualiza a sua reserva de dinheiro
- A interface pede a bebida à dispensa
- A dispensa envia a bebida seleccionada para o painel de interface.
- A interface devolve a bebida ao cliente

Na sequência destes dois passos fundamentais (identificação dos objectos envolvidos e identificação da sequência de acções) desenham-se com facilidade os respectivos diagramas de colaboração (ver Figuras 7.12 e 7.13). Repare-se que estes diagramas ajudam a clarificar interacções que normalmente não são tão evidentes especificadas de forma textual.

Figura 7.13: Diagrama de colaboração do Exemplo 7.3.

7.4 Diagramas de Estados

Um **diagrama de estados** (*statechart*), também conhecido por diagrama de transição de estado ou por máquina de estados, permite modelar o comportamento interno de um determinado objecto, subsistema ou sistema global.

Estes diagramas representam os possíveis estados de um objecto, as correspondentes transições entre estados, os eventos que fazem desencadear as transições, e as operações (acções e actividades) que são executadas dentro de um estado ou durante uma transição. Os objectos evoluem ao longo do tempo através de um conjunto de estados como resposta a eventos e à passagem de tempo.

Figura 7.14: Exemplo genérico de diagrama de estados.

A Figura 7.14 ilustra um diagrama de estados genérico, com um estado inicial, o estado X e um estado final. Os estados são representados por rectângulos com os cantos arredondados, excepto o estado inicial e final que têm ícones particulares. As transições são representadas graficamente por uma linha a cheio dirigida. A figura ilustra a visão detalhada de um estado, que para além do nome, pode ainda incluir um segundo compartimento com as acções e actividades realizadas.

Podem-se pensar em inúmeras situações, mais práticas ou mais conceptuais, de exemplos de sistemas ou de objectos que evoluem ao longo de distintos estados. Por exemplo:

(1) Uma lâmpada: que evolui entre os estados “acesa” e “apagada”, conforme se liga e desliga um interruptor (ver Figura 7.15).

(2) Uma máquina de lavar roupa: depois da passagem de um determinado período de tempo, a máquina de lavar termina o seu programa de lavagem, e inicia o de secagem.

(3) Uma instância da classe `javax.servlet.http.Servlet`: evolui ao longo de diferentes estados, tais como: em carregamento, inicializada, preparada para tratar pedido, destruída.

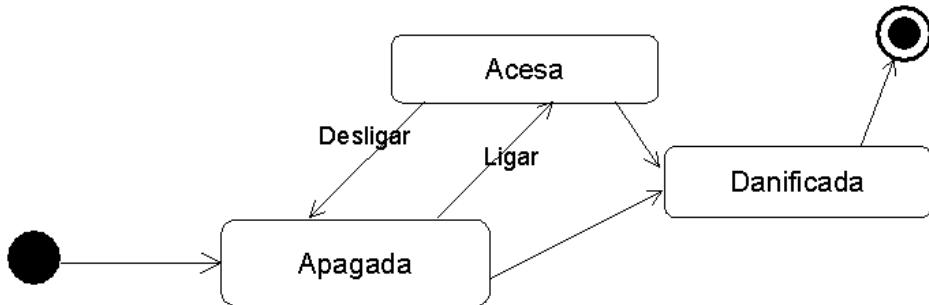



Figura 7.15: Diagrama de estados de uma lâmpada.

7.4.1 Estados

Um **estado** é uma situação registada por um objecto durante o seu respectivo ciclo de vida, durante a qual uma condição é verificada, vai executando alguma actividade, ou simplesmente espera que determinado evento ocorra.

Um estado tem diferentes partes, designadamente:

- Nome: Uma *string* que distingue o estado de outros estados; um estado sem nome designa-se por "estado anónimo".
- Acções de entrada e de saída: Acções executadas, respectivamente, no início (à entrada) ou no fim (à saída) do estado.
- Transições internas: transições que ocorrem mas que não alteram a mudança de estado.
- Sub-estados: uma estrutura aninhada de um estado, envolvendo sub-estados disjuntos (i.e., sequencialmente activos) ou concorrentes (i.e., concorrentemente activos).
- Eventos deferidos: uma lista de eventos que não são tratados no estado corrente, mas tratados pelo objecto num seu outro estado.

7.4.2 Transições

Uma **transição** é uma relação entre dois estados que especifica que um objecto que se encontre no primeiro estado, realizará um conjunto de acções e mudará para o segundo estado quando um determinado

evento ocorrer e determinadas condições se verificarem. Por exemplo, uma lâmpada pode transitar do estado “acesa” para o estado “apagada” quando o evento “desligar” ocorrer (ver Figura 7.15).

Uma transição é descrita integralmente pela seguinte sintaxe:

```
evento [condição com guarda] / acção
```

Contudo, podem existir transições com ou sem eventos, com ou sem condições com guarda, e com ou sem acções.

Uma transição tem diferentes partes, designadamente:

- O estado de origem e de destino, que a transição interliga.
- Evento de gatilho (*event trigger*): é o evento cuja recepção pelo objecto no estado origem proporciona a realização da transição, caso a condição de guarda seja satisfeita.
- Condição de guarda: expressão lógica que é avaliada quando a transição é lançada pela recepção do evento de gatilho. Se a expressão for avaliada verdadeira a transição ocorre, caso contrário a transição não ocorre e o evento é perdido.
- Acção (ver mais à frente nesta secção).

É possível existirem situações de transições sem gatilho (*triggerless*), i.e. transições sem eventos de gatilho: são transições que ocorrem apenas porque o estado origem termina a sua actividade normalmente.

Uma transição pode ter múltiplos estados-origem (representando a junção de múltiplos estados concorrentes) bem como múltiplos estados-destino (representando a divisão para múltiplos estados concorrentes), não sendo no entanto comum este tipo de utilização.

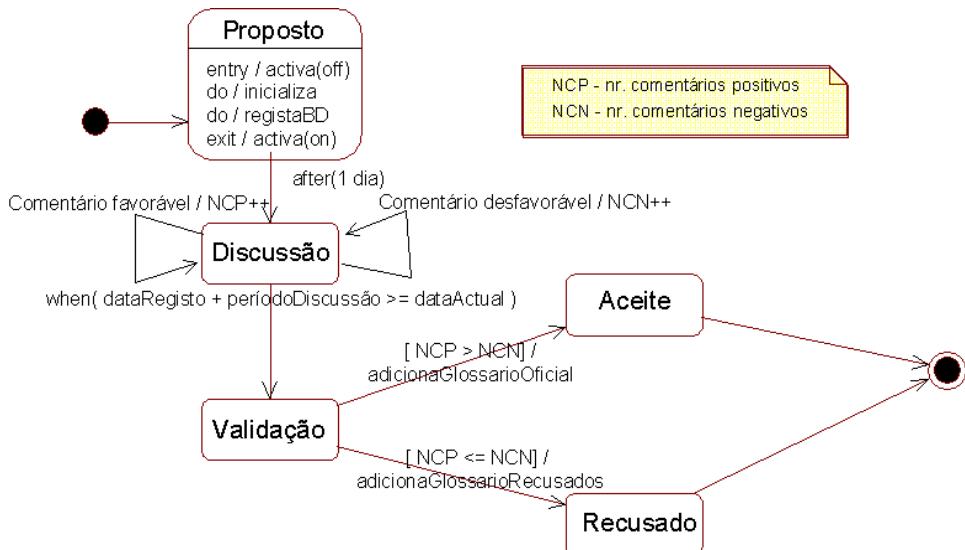



Figura 7.16: Diagrama de estados da classe `Termo`.

A Figura 7.16 ilustra vários tipos de transições representados no diagrama de estados das instâncias da classe `Termo`. Esta classe é usada no contexto de um sistema de gestão de glossário de termos técnicos informáticos (GTTI). O objectivo do sistema GTTI é permitir num contexto restrito de participantes, a construção de um glossário de forma colaborativa. Cada membro deste sistema pode propor termos (basicamente um termo em inglês e a sua respectiva tradução para português), os quais ficam pendentes durante um determinado período de discussão. Nesse período surgem comentários positivos e negativos de outros membros, no fim do qual segue-se um período de validação que tem como consequência aceitar ou recusar o termo proposto.

7.4.3 Eventos

Um **evento** é uma ocorrência de um estímulo que pode corresponder a uma transição de estado. Existem quatro tipos de eventos:

- Sinais
- Invocação
- Passagem de tempo

- Mudança de estado

Um **sinal** representa um objecto com nome que é enviado (lançado) assincronamente por um objecto e recebido (apanhado) por outro. O mecanismo de excepções suportado pela generalidade das modernas linguagens de programação (e.g., Java, C++, ObjectPascal) são os exemplos mais usuais de sinais.

Um **evento de invocação** corresponde ao lançamento de uma operação, tipicamente de modo síncrono.

Quer os eventos de invocação quer os sinais são representados graficamente da mesma forma (e.g., evento de invocação "`Comentário favorável`" da Figura 7.16). No entanto, a diferença deve ser clara a partir da informação implícita do modelo: em geral, um sinal é tratado pela própria máquina de estados, enquanto que evento de invocação por um método. Os sinais são trocados assincronamente entre objectos, o que implica que estes mantêm fluxos de actividades autónomos e concorrentes; por outro lado, os eventos de invocação ocorrem entre um ou mais objectos, com comunicação síncrona, o que implica que os objectos intervenientes são passivos no sentido que a sua execução ocorre sobre o mesmo fluxo de actividade.

O **evento de passagem de tempo** representa simplesmente o evento (natural e conhecido!) da "passagem de tempo". Normalmente não se especifica explicitamente este tipo de evento. Contudo, caso seja relevante explicitar e caracterizar tal evento, pode-se usar a palavra-chave `after` seguida por uma expressão que avalia o período de tempo (e.g. `after (1 dia)` da Figura 7.16). Outros exemplos de modelação de eventos de passagem de tempo: `after (1 hora)`; `after (1 segundo após a saída do estado Aceso)`.

O **evento de mudança de estado** representa uma mudança de estado ou a satisfação de uma determinada condição. Modeliza-se este tipo de evento usando a palavra-chave `when` seguida por uma determinada expressão lógica. Outros exemplos de modelação de eventos de mudança de estado: `when time > 12:29`; `when (nrPropostas > 3)`; `when(dataRegisto + períodoDiscussão >= dataActual)`.

7.4.4 Acções e Actividades

Uma **acção** é uma computação atómica, ou seja, em que se assume que a sua execução se realiza num período de tempo instantâneo e que é não interrompível. Acções podem consistir em invocação de métodos (tipicamente sobre o próprio objecto da máquina de estados), criação ou destruição de outro objecto, ou o envio de um sinal para outro objecto. Num estado podem especificar-se acções de entrada, prefixadas com a palavra chave `entry`; acções de saída, prefixadas com a palavra chave `exit`; e ainda acções relativas a eventos desencadeados e tratados dentro do estado (neste caso usa-se a notação `evento / acção`).

Ao contrário, uma **actividade** é uma computação não atómica, i.e. que pode ser interrompível por outros eventos. Corresponde normalmente a uma operação complexa, eventualmente descrita por um outro diagrama de estados incluso. As actividades são elementos básicos dos diagramas de actividades (ver Secção 7.5) mas também podem ser referidas na especificação de um estado, sendo nesse caso prefixadas com a palavra chave `do` (ver Figura 7.17).

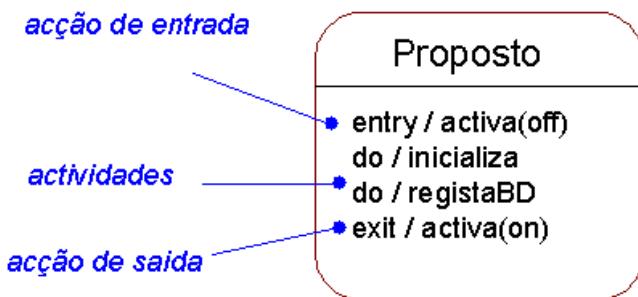



Figura 7.17: Especificação de acções e actividades num estado.

Pode-se ainda especificar uma sequência de acções realizáveis num determinado estado (e.g., `do / operação1();operação2();operação3()`).

7.4.5 Sub-Estados

Um **sub-estado** é um conceito avançado dos diagramas de estados do UML. Um sub-estado é um estado que se encontra definido dentro de outro (super)estado. A ideia subjacente ao conceito de sub-estado é a abstracção: uma máquina de estados pode ser descrita com diferentes níveis de abstracção e de detalhe conforme seja necessário ou relevante em cada momento.

Um estado que tenha um conjunto de sub-estados mais detalhados designa-se por "estado composto". Um **estado composto** pode conter quer sub-estados concorrentes (ortogonais) quer sub-estados sequenciais (disjuntos). Um estado pode ser decomposto em vários níveis de inclusão.

O Exemplo 7.4 ilustra uma decomposição de estados em sub-estados.

Exemplo 7.4: Diagrama de Estados de um PC.

A Figura 7.18 dá uma visão simplificada do ciclo de vida de um PC que evolui ao longo de três estados sequenciais: em "inicialização", "trabalhando" e "fecho". A Figura 7.19 introduz uma primeira variante ao exemplo base considerando que no PC se encontra instalado um *screen saver*, que é activado após um determinado tempo de inactividade do acesso ao disco ou de operações de entrada/saída.

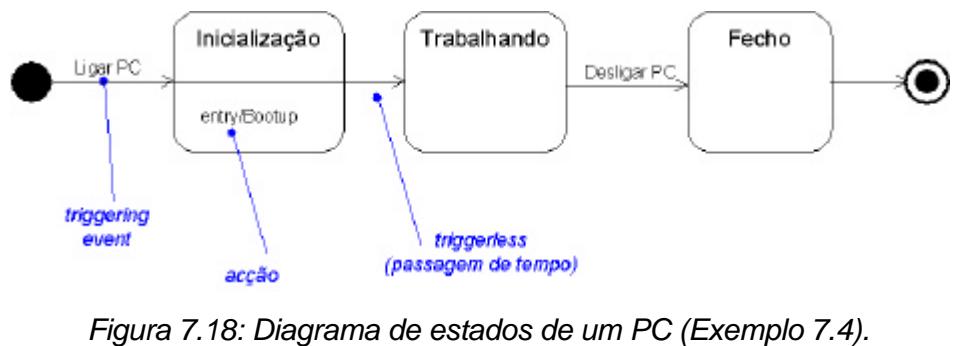



Figura 7.18: Diagrama de estados de um PC (Exemplo 7.4).

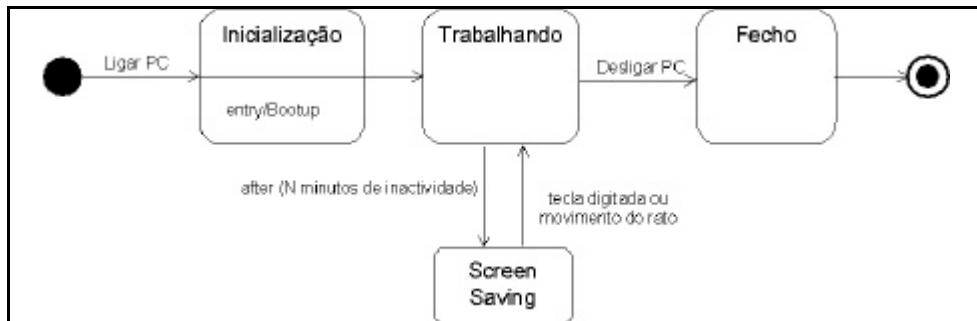



Figura 7.19: Diagrama de estados de um PC, variante 1 (Exemplo 7.4).

A Figura 7.20 detalha o estado "Trabalhando" tendo em conta a realização concorrente de duas sub-máquinas de estado. Uma tem a ver com o mecanismo de tratamento de eventos conduzidos pelo utilizador (e.g., via teclado ou rato): o PC espera pelo *input* do utilizador, seguidamente regista esse *input* e por fim faz a respectiva visualização no monitor. A outra máquina de estados tem a ver com a leitura do tempo do relógio do PC e a correspondente actualização no seu monitor. Ambas as maquinas de estados actuam concorrentemente de forma independente entre si.

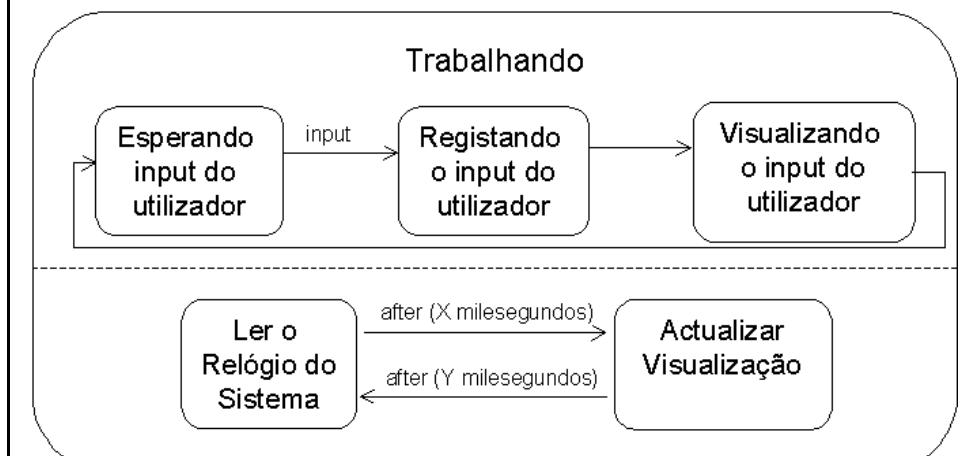



Figura 7.20: Diagrama de estados de um PC, variante 2 (Exemplo 7.4).

7.5 Diagramas de Actividades

Um **diagrama de actividades** é um caso particular de um diagrama de estados, no qual todos ou a maioria dos estados são “estados de actividades” e todas ou a maioria das transições são desencadeadas pela conclusão das actividades dos estados anteriores.

Ambos os tipos de diagramas são utilizados para modelar o tempo de vida de um objecto ou sistema. Contudo, um diagrama de actividades ilustra o fluxo de controlo entre actividades, enquanto que um diagrama de estados ilustra o fluxo de controlo entre estados. Por outro lado, os diagramas de interacção ilustram fluxos de controlo entre objectos. Enquanto nos diagramas de interacção o foco é na visualização das mensagens trocadas entre objectos, nos diagramas de actividades a atenção incide na visualização das operações realizadas pelos objectos intervenientes.

Uma actividade, como visto na Secção 7.4.4, corresponde a uma execução não atómica dentro de uma máquina de estados, ou doutra forma, corresponde à execução de um conjunto de acções.

Os diagramas de actividades correspondem aos conhecidos “fluxogramas”. Fornecem uma visão simplificada do fluxo de controlo de uma **operação** ou de um **processo de negócio**, também designado por “*workflow*”.

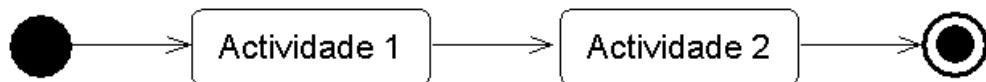



Figura 7.21: Exemplo genérico de diagrama de actividades.

A Figura 7.21 ilustra um exemplo genérico de um diagrama de actividades.

Estes diagramas contêm genericamente:

- Estados-acção: execuções atómicas, não interrompíveis, com tempo de execução irrelevante.
- Estados-actividade: execuções não atómicas (decompostas), interrompíveis, em que o tempo de execução é normalmente relevante.
- Transições.

- Objectos.

Conforme ilustrado na Figura 7.22 não existe uma distinção na representação gráfica entre estados-acção e estados-actividade, excepto que os estados-actividade podem conter partes adicionais, tais como especificação de acções de entrada e de saída e sub-máquinas de estado. De ora em diante utilizar-se-á, por motivos de simplicidade de leitura do texto, o termo “actividade” para representar qualquer dos estados referidos.

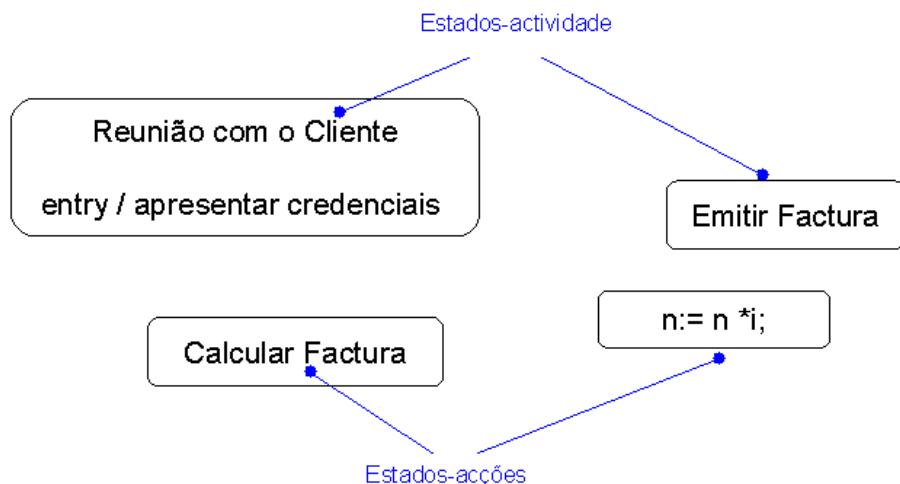



Figura 7.22: Estados-actividade versus estados-acção.

No desenho de diagramas de actividades (tal como nos diagramas de estado) é comum a utilização de um conjunto de mecanismos que de seguida se detalham, designadamente a especificação de tomada de decisão, de concorrência (i.e., a execução de actividades concorrentes), de troca de sinais entre diferentes máquinas de estado ou *workflow*, e interacção com objectos.

7.5.1 Decisões

A **tomada de decisão** é um mecanismo comum no desenho de diagramas de actividades (e de estado), que consiste em especificar que actividade deve ser realizada após a execução da actividade

corrente. Tal especificação é suportada por uma condição com guarda (i.e., expressão lógica) que é colocada de forma adjacente à transição de estado correspondente.

A figura 7.23 ilustra dois esquemas alternativos para a representação da tomada de decisão relativamente ao processo "levantar da cama". Os esquemas são semanticamente equivalentes e representam o seguinte fluxo de actividades: a seguir à actividade "Acordar" alguém pode decidir "Tomar pequeno-almoço" ou "Voltar para a cama"… mas não ambas as actividades.

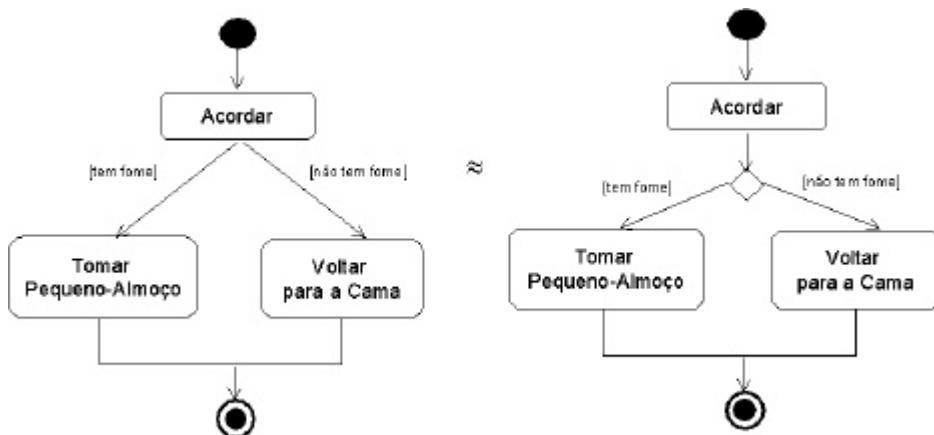
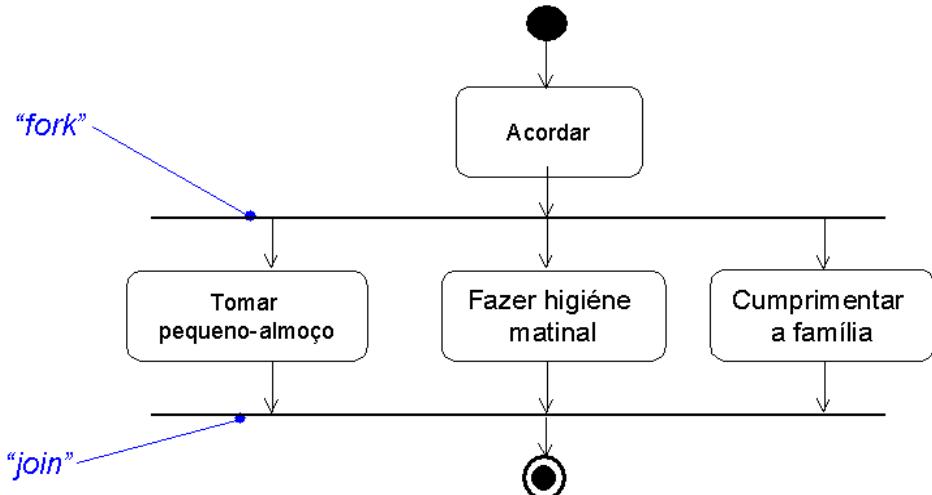



Figura 7.23: Decisão em diagramas de actividades.

Note-se que as transições entre as actividades referidas são desencadeadas apenas na circunstância de uma das condições com guardas ser satisfeita, no exemplo ilustrado pelo facto de "ter ou não fome".

7.5.2 Caminhos Concorrentes

Considere-se que o processo de "levantar da cama" implica a execução das seguintes actividades "tomar pequeno-almoço", "fazer a higiene matinal" e "cumprimentar a família". Considere-se que essas actividades têm de se realizar obrigatoriamente, embora não seja relevante a sua ordem de execução.

O problema colocado representa uma situação típica na modelação de *workflows*: representar a execução independente e concorrente de um conjunto de actividades.

Figura 7.24: Concorrência em diagramas de actividades.

O UML providencia a solução a esta questão através dos conceitos de **difusão** (*fork*) e de **junção** (*join*) de actividades, representados graficamente por linhas a cheio conforme ilustrado na Figura 7.24. Esta linha designa-se por **barra de sincronização**.

Podem-se representar hierarquias de fluxos de actividades concorrentes, i.e., podem-se representar actividades e pares de linhas de difusão/junção dentro de outros pares de linhas de difusão/junção. Todavia, o número de linhas de difusão e de junção deve ser balanceado correspondentemente.

7.5.3 Pistas (Swimlanes)

Na modelação de processos de negócio é comum a realização de actividades por várias entidades, participantes no dito processo. O UML propõe o conceito de **pistas** (***swimlanes***) como elemento que permite agrupar as várias actividades da responsabilidade de cada entidade participante. Cada grupo é separado por uma linha vertical.

Cada pista tem um nome único dentro do seu diagrama, que deve corresponder ao nome da entidade participante, a qual deve ser uma entidade do mundo real. Por exemplo, o nome de um perfil de utilizador, o nome de uma organização, ou o nome de um sistema de informação.

A Figura 7.25 representa o processo de negócio “Preparar Proposta”, típico de uma empresa de serviços. São representadas três entidades participantes: o cliente, que solicita a proposta; o gestor comercial, que prepara o orçamento; e o gestor de produção, que pode eventualmente intervir no processo, caso o serviço solicitado exija aspectos específicos da produção.

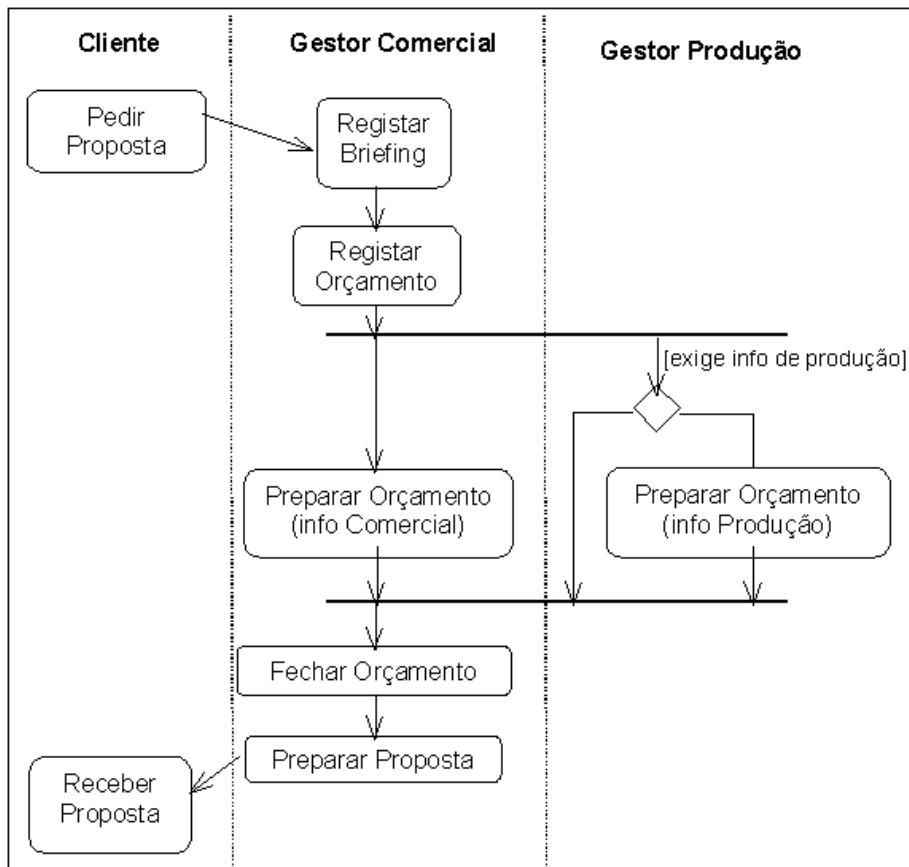



Figura 7.25: Diagramas de actividades com pistas.

Num diagrama de actividades com pistas, cada actividade pertence em geral a uma única pista, apenas as transições cruzam as pistas.

Contudo, podem existir situações em que uma actividade é realizada por duas entidades. Nestes casos, a actividade é representada sobre a linha separadora em comum. Claro que esta solução não é generalizável: como representar actividades realizadas por mais que três entidades?; ou como representar actividades realizadas por duas entidades que não se encontram graficamente adjacentes? (Este aspecto é uma limitação actual do UML, não tratado na especificação 1.3, existindo contudo algumas soluções que poderão vir a ser adoptadas numa sua versão futura.)

7.5.4 Actividades e Objectos

Os diagramas de actividades podem explicitar relações entre actividades e objectos. Por exemplo, no diagrama da Figura 7.26 incluíram-se relações entre actividades e uma instância da classe `Orçamento`. Estas relações de dependência permitem ilustrar o fluxo de um objecto ao longo de um conjunto de actividades, representadas por linhas dirigidas a tracejado.

Para além de se ilustrar o fluxo de um objecto num diagrama de actividades, podem ainda ilustrar-se os seus papéis, atributos e estado. No exemplo da Figura 7.26 apresenta-se o estado do objecto `:Orçamento` entre parêntesis rectos, que ao longo do tempo se encontra em “aberto”, “preparado” ou “fechado”.

Cliente
Gestor Comercial
Gestor Produção
Pedir Proposta
Registar Briefing
:Orçamento [aberto]
Registar Orçamento
[exige info de produção]
:Orçamento [preparação]
Preparar Orçamento (info Comercial)
Preparar Orçamento (info Produção)
Fechar Orçamento
:Orçamento [fechado]
Preparar Proposta
Receber Proposta

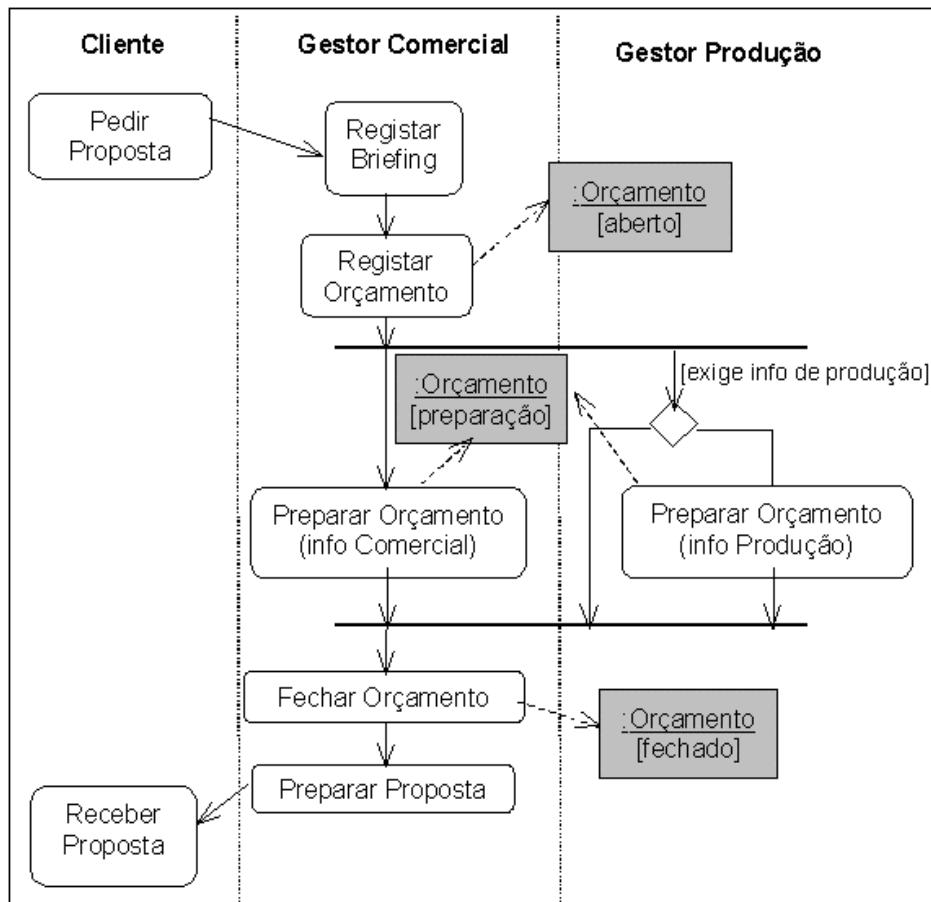



Figura 7.26: Diagramas de actividades com objectos.

7.5.5 Envio e Recepção de Sinais

O UML adopta os conceitos de **evento de envio** (*output event*) e **evento de recepção** (*input event*) de sinais, os quais são usados, respectivamente para explicitar a emissão e a recepção de um determinado sinal. Não sendo obrigatório, a sua utilização permite, contudo, explicitar:

- Relações de dependência (note-se a seta dirigida a tracejado entre os eventos de envio e de recepção do sinal) entre distintas máquinas de estado ou distintos diagramas de actividades pela troca de eventos assíncronos

- Dentro da mesma máquina de estados, eventos que deverão ocorrer durante as respectivas transições de estados/actividades.

O evento de emissão é representado graficamente por um polígono convexo, enquanto que o evento de recepção é representado por um polígono côncavo.

A Figura 7.27 ilustra a emissão e recepção do sinal "mudar(canal)" entre os diagramas de actividades correspondentes ao controlo de uma televisão.

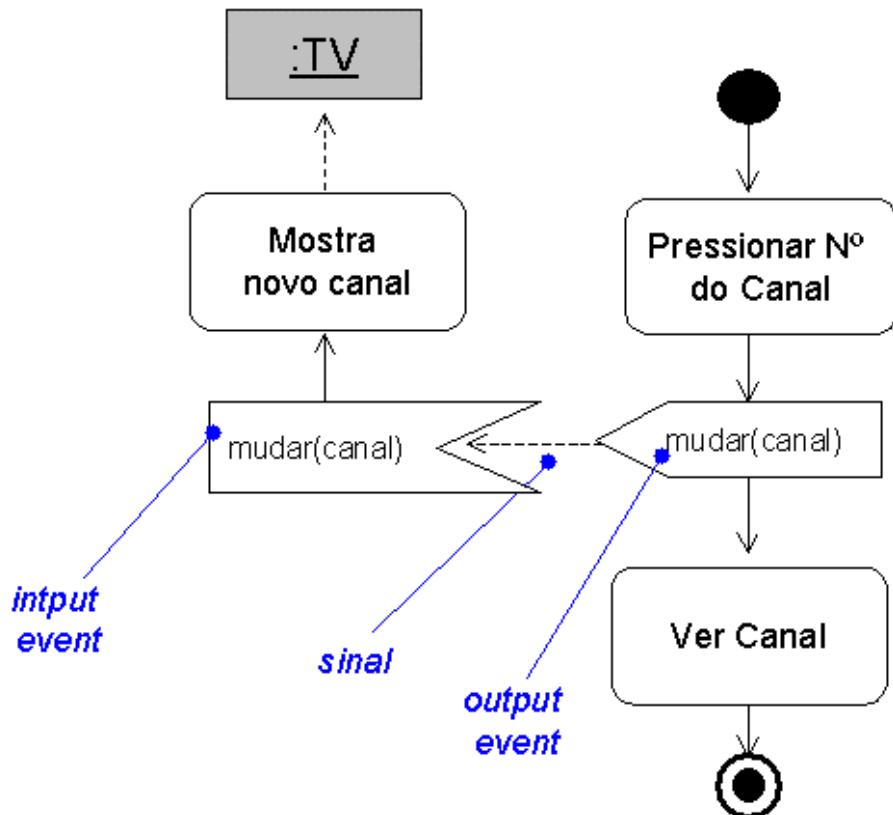



Figura 7.27: Diagramas de actividades com símbolos para envio e recepção de sinais.

7.5.6 Utilizações Típicas

Os diagramas de actividades são utilizados para modelar os aspectos de comportamento de um sistema como um todo, de um subsistema, de uma operação ou de uma classe. Pode-se também associar um diagrama de actividades a um caso de utilização ou a uma colaboração (para modelar o comportamento de um conjunto de objectos).

No entanto, os diagramas de actividades são usados principalmente nas duas situações seguintes:

- Para especificar **operações**: Neste caso os diagramas de actividades são usados como fluxogramas para especificar detalhadamente um algoritmo. São particularmente usados os conceitos de tomada de decisão, de difusão (*fork*) e de junção (*join*).
- Para especificar ***workflows*** ou **processos de negócio**: Neste caso o foco dos diagramas de actividades reside na identificação dos actores intervenientes e a correspondente colaboração com o sistema. São particularmente usados os conceitos das pistas e da modelação do fluxo de objectos.

Apresentam-se de seguida dois exemplos que clarificam a aplicação dos diagramas de actividades.

Exemplo 7.5: Diagrama de actividades da operação de Fibonacci.

Considere a função Fibonacci no domínio dos números inteiros dada pela fórmula:

```
fib(n) =1, se n ≤ 2; = fib(n-1)+ fib(n-2), se n> 2
```

A Figura 7.28 ilustra o diagrama de actividades (neste caso, tomando a forma de fluxograma) correspondente ao algoritmo de implementação da dita função. Neste diagrama é especificado um dado algoritmo, mas outras variantes deste poderiam ser especificados. Por exemplo, poder-se-ia apenas realizar um teste (n ≤ 2) no início, em vez de dois (n=1 e n=2), etc. (Fica como exercício o desenho de outros algoritmos alternativos.) Note-se como se especifica iterações à custa de tomada de decisões, resultando graficamente em fluxos cíclicos.

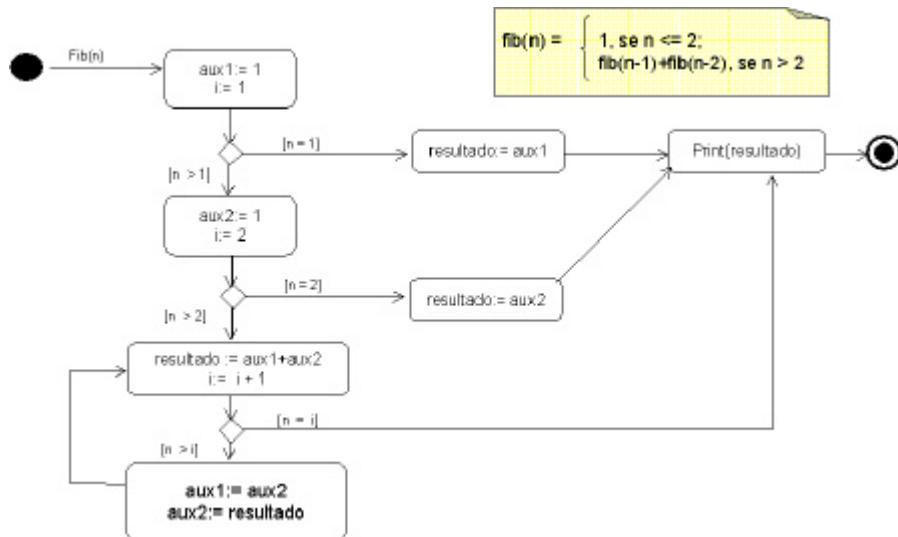



Figura 7.28: Diagrama de actividades da operação de Fibonacci (Exemplo 7.5).

Exemplo 7.6: Diagrama de actividades da disseminação de eventos no WebDEI.

Considere o sistema WebDEI, que é um sistema de informação de um hipotético departamento de engenharia informática (DEI), com interface Web. (Para mais detalhes sobre o WebDEI consulte-se as Secções 11.4 e 11.5.)

Entre outros processos e funcionalidades, o WebDEI providencia o processo de negócio designado por "disseminação automática de eventos". A ideia geral, é que os membros registados do WebDEI (em geral, professores) podem submeter anúncios de eventos que se venham a realizar a curto/médio prazo, e que poderão ser pertinentes para os restantes membros, ou eventualmente para o público em geral. Complementarmente ao processo de submissão de anúncios de eventos, o processo de disseminação dos ditos eventos submetidos é realizado com uma determinada periodicidade e consiste no envio, por e-mail ou SMS, de um sumário de todos os eventos relevantes, a todos os utilizadores registados que tenham manifestado o interesse em receber este tipo de informação.

A Figura 7.29 ilustra através de um diagrama de actividades o processo de negócio da "disseminação automática de eventos". Identificaram-se três intervenientes: o temporizador, que é o responsável por desencadear periodicamente deste processo; o próprio sistema WebDEI, responsável pela generalidade das actividades do processo; e o utilizador registado, que recebe o sumário dos eventos relevantes para período em causa.

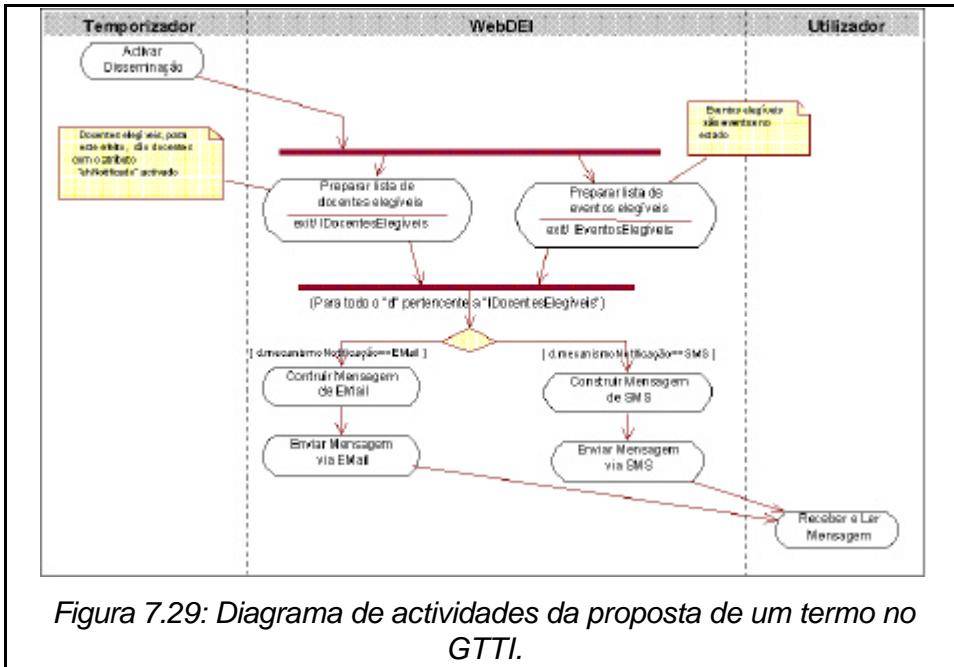



Figura 7.29: Diagrama de actividades da proposta de um termo no GTTI.

7.6 Exercícios

Ex.48. Considere o melhor cenário para o caso de utilização "Enviar Fax" (o cenário em que tudo corre bem"). Considere um sistema composto pelos seguintes objectos: máquina que envia; máquina que recebe; uma central que encaminha faxes e chamadas telefónicas. Desenhe o diagrama de sequência respectivo.

Ex.49. Considerem-se outros cenários para o caso de utilização "Comprar Bebida" relativo ao sistema "Máquina de Bebidas" introduzido anteriormente:

- O utilizador introduziu mais dinheiro que o valor da bebida, e a máquina tem dinheiro para troco
- O utilizador introduziu mais dinheiro que o valor da bebida, e a máquina não tem dinheiro para troco

Desenhe os respectivos diagramas de sequência e de colaboração.

Ex.50. Desenhe o diagrama de estados de uma tostadeira. Defina os diferentes estados do pão na tostadeira, sem esquecer de especificar os necessários eventos, acções, e condições com guarda.

Ex.51. Desenhe o diagrama detalhado do estado "*Screen Saving* de um PC" que inclua sub-estados concorrentes (ver Exemplo 7.4). Considere, por exemplo, os estados responsáveis por tratarem o *input* do utilizador, outros responsáveis pela geração de imagens e actualização dinâmica no monitor.

Ex.52. Desenhe o diagrama de estados da classe `javax.servlet.http.Servlet`. Considere que um *servlet* Java evolui ao longo de diferentes estados, tais como: carregamento, inicialização, tratar pedido, destruição.

Ex.53. Idem ao exercício anterior relativamente à classe `java.applet.Applet`.

Ex.54. Desenhe o diagrama de actividades correspondente ao algoritmo do factorial de "n" ($n! = 1$ se $n \leq 1$; $n*(n-1)!$ se $n > 1$).

Ex.55. Desenhe o diagrama de actividades correspondente ao seguinte processo de negócio: "gestão de encontros com clientes":

1. Um vendedor telefona ao cliente e marca uma reunião.
2. Se a reunião é na empresa, os técnicos da empresa preparam a sala de conferências para a apresentação.
3. Se a reunião é fora da empresa (no escritório do cliente) um consultor prepara a apresentação num computador portátil.
4. O consultor e o vendedor reúnem-se com o cliente no local e hora combinada.
5. O vendedor envia ao cliente uma carta a resumir o "sucesso" da reunião.
6. Se a reunião resultou na identificação de um problema, o consultor escreve uma proposta e envia-a para o cliente.

Ex.56. Modifique o diagrama de actividades da Figura 7.24 de modo a especificar o processo "levantar da cama" com as seguintes considerações. A seguir à actividade "acordar" um indivíduo realiza geralmente as seguintes actividades, sem uma ordem predefinida: "tomar pequeno-almoço", "fazer a higiene matinal" e "cumprimentar a família". Contudo, (1) apenas toma o pequeno-almoço se não tiver pressa; e (2) apenas cumprimenta a família se estiver bem disposto.

Ex.57. Considere o seguinte código Java constituído pelas classes `SimpleThread` e `TwoThreadsTest`. Desenhe o diagrama de classes que o suporta e o diagrama de colaboração correspondente a instâncias da classe `TwoThreadsTest`.

```
public class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
            sleep((long)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
}
public class TwoThreadsTest {
  public static void main (String[] args) {
    SimpleThread jamaica, fiji;
    jamaica= new SimpleThread("Jamaica");
    fiji= new SimpleThread("Fiji")
    jamaica.start();
    fiji.start();
  }
}
```

Capítulo 8 - UML – MODELAÇÃO DA ARQUITECTURA

Tópicos

- Introdução
- Componentes e Nós
- Diagramas de Componentes
- Diagramas de Instalação
- Exercícios

8.1 Introdução

Os **diagramas de arquitectura** (ou **diagramas de implementação**)[1] descrevem aspectos da fase de implementação e instalação de um sistema de software, designadamente a estrutura e dependências de código fonte e de módulos executáveis tal como a sua respectiva instalação nas diferentes plataformas compu-tacionais subjacentes.

---

[1] A especificação 1.3 do UML designa-os por "diagramas de implementação". Todavia, a designação de "diagramas de arquitectura" parece-nos mais adequada tendo em conta que estes diagramas podem ser aplicados no desenho de diferentes tipos de arquitecturas. Por exemplo: arquitectura de sistemas de informação (que é a sua aplicação mais conhecida) ou arquitectura de negócios e de organizações.

Estes diagramas apresentam-se sob duas formas: diagramas de componentes e diagramas de instalação.

Os diagramas de componentes são usados para modelar a arquitectura de um sistema de software na perspectiva dos seus componentes digitais (e.g., ficheiros de código fonte, de executáveis, de configuração, tabelas de dados, documentos de gestão do projecto), explicitando principalmente as suas múltiplas dependências.

Os diagramas de instalação, por outro lado, são usados para modelar a arquitectura de um sistema informático na perspectiva dos seus componentes físicos (e.g., computadores, adaptadores de rede, impressoras, *routers*, cablagem), explicitando as suas dependências de comunicação. Permitem ainda identificar quais os componentes que são instalados em cada nó computacional.

Estes diagramas podem também ser aplicados na modelação de negócios e de organizações caso se considere que os componentes digitais sejam procedimentos e regras de negócio e que os componentes não digitais (i.e., os nós) constituam a infra-estrutura da organização através de um conjunto de recursos (humanos e outros) do negócio.

(Na nossa opinião os diagramas de implementação constituem a parte mais limitada, mal explorada e compreendida do UML. Há inúmeros aspectos de desenho e de organização que a sua actual versão não aborda, deixando-os em aberto, ao critério que os seus utilizadores venham a definir, caso a caso. No lado oposto desta abordagem de flexibilidade e de não definição, encontra-se, por exemplo, o EAB (*Entreprise Application Blueprint*) [Boar98] que propõe uma notação muito completa e rigorosa para desenho de arquitecturas de sistemas de informação, em particular para o desenho de sistemas, plataformas e suas inter-relações.)

8.2 Componentes e Nós

8.2.1 Componentes

Um **componente** é uma peça básica na implementação de um sistema; consiste, na prática, num conjunto de artefactos físicos em formato

digital, por exemplo ficheiros de código (fonte, binário ou executáveis) ou ficheiros de documentos relativos ao negócio.

Definem-se pelo menos três tipos distintos de componentes:

- **Componentes de instalação**: constituem a base dos sistemas executáveis (e.g., DLL, executáveis, controlos Active-X, classes Java).

- **Componentes de trabalho**: a partir dos quais são criados os componentes de instalação (e.g., ficheiros com código fonte, ficheiros de dados, documentos).
- **Componentes de execução**: criados como resultado da execução de um sistema (e.g., processos, *threads*, agentes de software). Estes componentes são apenas representados nos diagramas de instalação (ver mais à frente).

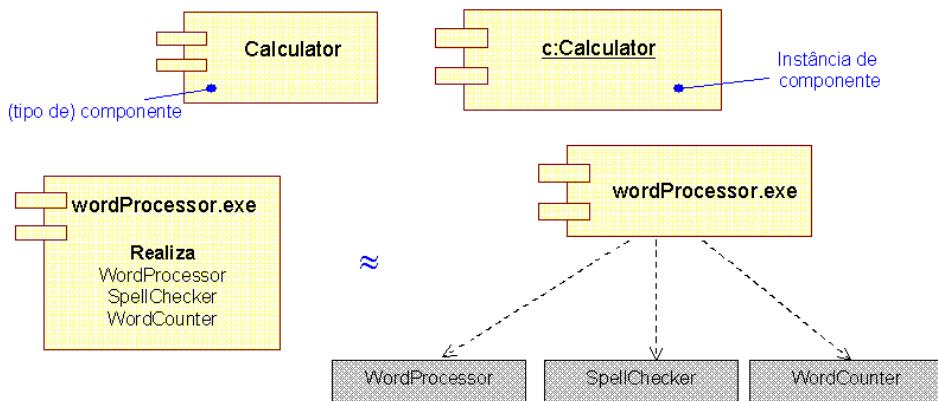



Figura 8.1: Representação gráfica de componentes.

Um componente de software é uma parte física de um sistema: existe de facto num determinado computador e não apenas na mente do analista, como acontece com o conceito de classe. Adicionalmente, um componente implementa uma ou mais classes, as quais são representadas dentro do ícone de componente ou com relações explícitas de dependência de implementação, conforme ilustrado na Figura 8.1.

Elementos que residam num componente são apresentados dentro do símbolo do componente respectivo. Pode-se, se for conveniente, representar o tipo de visibilidade desses elementos à semelhança do que acontece com os pacotes. O significado da visibilidade depende do tipo de componente. Por exemplo, se for um componente com código fonte, pode corresponder a controlar a acessibilidade aos seus construtores internos; se for um componente com código executável, a visibilidade pode corresponder à possibilidade de outros componentes executáveis poderem aceder ou invocar o seu próprio código.

O desenvolvimento de software baseado em componentes pressupõe a existência de componentes com um sentido mais restrito que este adoptado no UML. Ou seja, a noção de "componente" em UML é lata, mas inclui naturalmente a noção de componente de software encontrada por exemplo nas propostas do Java Beans, Enterprise Java Beans, ou Active-X. Um aspecto importante ligado à noção de componente tem a ver, como é analisado na Secção 6.4, com a noção de interface. Tipicamente, os componentes de software (no sentido restrito acima referido) implementam uma ou mais interfaces e é através destas interfaces que providenciam as suas funcionalidades a outros componentes. A Figura 8.2 ilustra a relação (de realização) entre componentes e interfaces, tal como a relação de dependência entre componentes, que se faz através do conceito de interface.

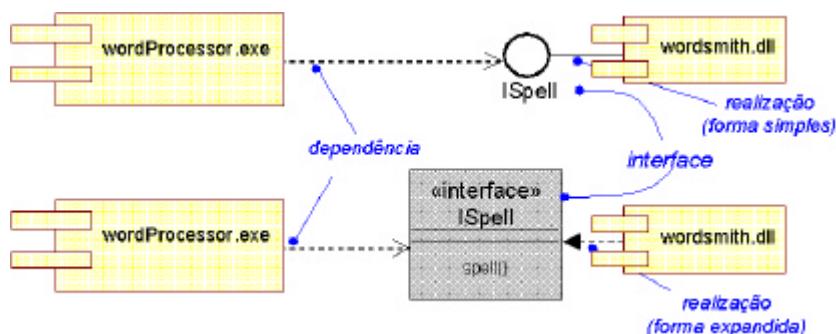



Figura 8.2: Componentes e Interfaces.

A especificação 1.3 do UML identifica os seguintes estereótipos para componentes:

- «`document`»: denota um documento.

- «`executable`»: denota um programa que possa ser executado num nó.
- «`file`»: denota um documento contendo código fonte ou dados.
- «`library`»: denota uma biblioteca dinâmica ou estática.
- «`table`»: denota uma tabela de uma base de dados.

8.2.2 Nós

Um **nó** é um objecto físico que representa um recurso de processamento, geralmente tendo capacidades de memória e de processamento. Os nós podem consistir em recursos computacionais (hardware), mas também em recursos humanos ou recursos de processamento mecânico. Os nós podem ser representados como tipos e como instâncias. Instâncias de nós podem conter instâncias de objectos e de componentes.

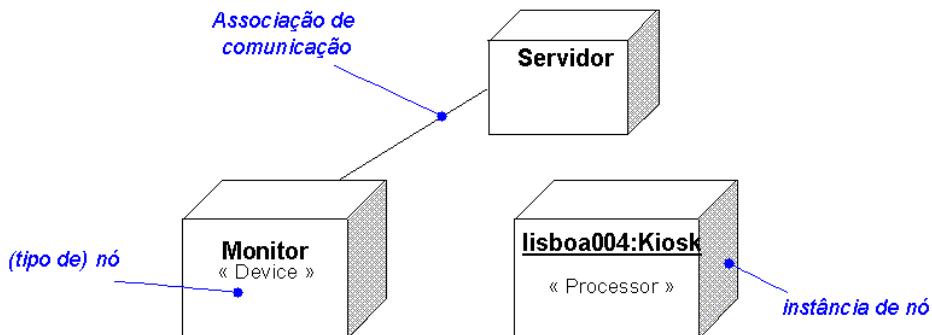



Figura 8.3: Representação gráfica de nós.

Um nó é representado como um cubo tridimensional conforme ilustrado na Figura 8.3. Dois nós podem-se encontrar ligados através de relações de associação. Estas especificam a existência de caminhos de comunicação entre os correspondentes nós e podem ser caracterizadas por um estereótipo, de modo a explicitar o tipo de comunicação envolvido (e.g., o tipo de canal ou o tipo de rede).

As propriedades dos nós (e.g., capacidade de memória principal, número de processadores, data de aquisição, …) são representadas por

marcas com valores. Por outro lado, podem-se definir estereótipos, com ícones correspondentes, para modelar diferentes tipos de recursos de processamento.

Para efeito dos exemplos descritos neste livro assume-se a existência de dois estereótipos de nós para representação de recursos computacionais:

- «`processor`»: denota um nó que pode executar um componente de software.
- «`device`»: denota um nó que não tem capacidade para executar componentes de software; e.g., uma impressora, um *scanner*, ou um monitor.

Assume-se, por omissão, que um nó é do estereótipo «`processor`» (e.g., nó `Servidor` da Figura 8.3).

8.2.3 Relações entre Nós e Componentes

Um nó pode conter componentes. Tal facto pode ser traduzido pela inclusão dos componentes no símbolo do nó, ou pelo estabelecimento de uma relação de dependência, de estereótipo «`support`» entre o nó e os componentes suportados, conforme ilustrado na Figura 8.4.

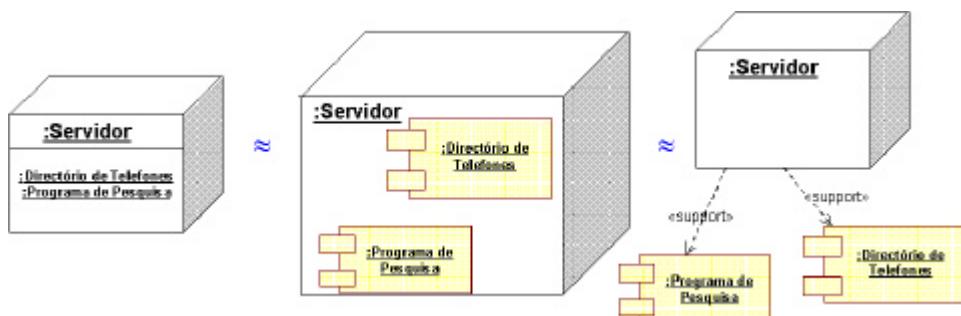



Figura 8.4: Relação entre nós e componentes.

Nós e componentes partilham um conjunto de semelhanças e diferenças. As **semelhanças** são que ambos podem (1) participar em relações de generalização, dependência e associação; (2) ser aninhados; (3) ter instâncias; e (4) participar em interacções. As **diferenças** são que os (1) componentes são elementos que participam

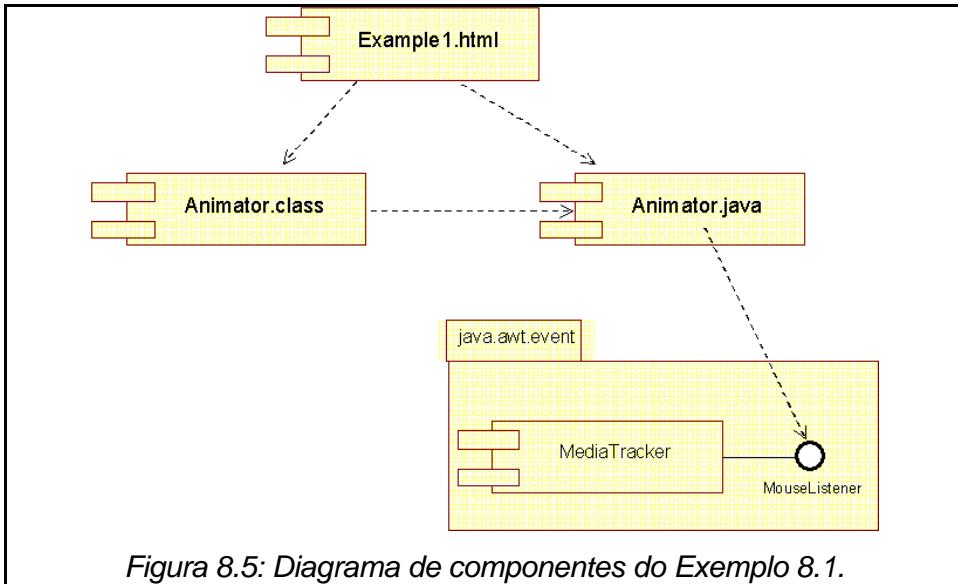
na execução de um sistema; nós são elementos que suportam e executam componentes; e que os (2) componentes representam agrupamento físico de elementos lógicos; nós representam a instalação física de componentes.

8.3 Diagramas de Componentes

Um **diagrama de componentes** ilustra as dependências entre vários componentes de software. Entre outros, incluem-se nesta definição lata: artefactos de código fonte, de código binário, de código executável, procedimentos de negócio e documentos. Um módulo de software pode ser representado por um estereótipo, por exemplo, para ter uma apresentação gráfica distinta de outros tipos de componentes.

Um diagrama de componentes representa apenas tipos de componentes e nunca instâncias de componentes. Para ilustrar instâncias de componentes deve ser usado um diagrama de instalação (possivelmente, uma versão simplificada sem nós).

Entre outras motivações para a construção de modelos de componentes, salientam-se as seguintes:

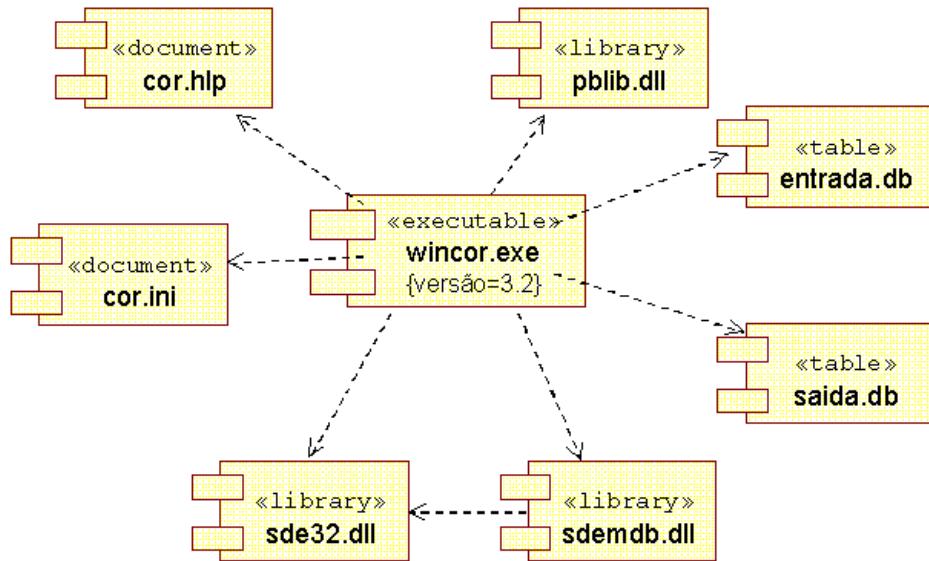
- Os clientes podem ver a estrutura final do sistema, mesmo antes deste estar concluído.
- A equipa de desenvolvimento tem uma visão da arquitectura física do sistema, pelo que pode trabalhar de forma mais controlada e sistemática.
- Os escritores técnicos (que produzem, por exemplo, a documentação do sistema, manuais de utilizador, manuais técnicos) podem entender melhor o que estão a escrever, detalhar alguns aspectos do sistema, mesmo antes de este se encontrar concluído.

Apresentam-se de seguida dois exemplos de aplicação de diagramas de componentes: um que ilustra as dependências de artefactos físicos referenciados numa página HTML; e um segundo exemplo que ilustra as dependências entre módulos de instalação constituintes de uma aplicação típica do Windows.

Exemplo 8.1: Diagrama de Componentes relativo a uma Página HTML.

Considere a página Web `Example1.html` com uma referência a um applet Java e com o seguinte conteúdo:

```
      <html>
        <head>
          <title>The   Animator   Applet   (1.1)   -   example
1</title>
        </head>
        <body>
            <h1>The Animator Applet (1.1) - example 1</h1>
            <applet codebase="." code=Animator.class
                                width=460 height=160>
          </applet>
         <a href="Animator.java">The source.</a>
            <hr>
        </body>
      </html>
```

O diagrama de componentes correspondente a este “mini-sistema” consiste nos seguintes ficheiros: `example1.html`, `Animator.class`, e `Animator.java`. A Figura 8.5 ilustra essas relações de dependência. Note-se em particular a relação de dependência explicita entre o componente `Animator.java` e a interface Java `MouseListener.java` definida no pacote `java.awt.event`.

Figura 8.5: Diagrama de componentes do Exemplo 8.1.

Exemplo 8.2: Diagrama de Componentes relativo à instalação de uma aplicação.

Considere a aplicação WinCOR desenvolvida sobre ambiente MS-Windows e responsável pela gestão de (entrada e saída de) correspondência de uma organização. A aplicação consiste num conjunto variado de componentes de instalação, nomeadamente:

- `wincor.exe`: ficheiro que contêm o executável da aplicação
- `pblib32.dll`, `sde32.dll`, `sdemdb32.dll`: bibliotecas com código binário que providenciam funcionalidades adicionais
- `wincor.hlp`: ficheiro de ajuda sobre a aplicação.
- `wincor.ini`: ficheiro de configuração da aplicação
- `entrada.db`, `saida.db`: ficheiros/tabelas da base de dados de suporte

A Figura 8.6 ilustra o respectivo diagrama de componentes para a situação descrita. Note-se nas dependências identificadas entre os diferentes componentes de instalação. Estas dependências referem que o executável `wincor.exe` (i.e., a aplicação WinCOR) apenas pode correr se todos os restantes componentes tiverem sido instalados

adequadamente e que o módulo `sdemdb32.dll` depende do módulo `sde32.dll`.

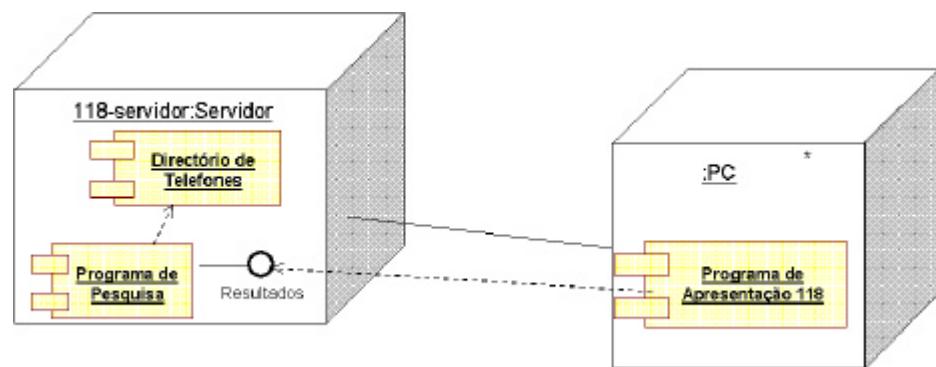
«document» cor.hlp
«library» pblib.dll
«table» entrada.db
«executable» wincor.exe {versão=3.2}
«document» cor.ini
«table» saida.db
«library» sde32.dll
«library» sdemdb.dll

Figura 8.6: Diagrama de componentes do Exemplo 8.2.

Neste exemplo houve o cuidado particular de se explicitar os estereótipos dos diferentes componentes envolvidos.

8.4 Diagramas de Instalação

Um diagrama de instalação ilustra a configuração dos elementos de processamento e dos componentes de software, processos e objectos neles suportados. Instâncias de componentes de software representam manifestações de execução das unidades de código.

Um **diagrama de instalação** (também designado nalgumas circunstâncias por **diagramas de distribuição**) consiste num conjunto de nós ligados por associações de comunicação. Os nós podem conter instâncias de componentes (de execução), o que significa que um

componente é instalado e executado num nó. Por outro lado, os componentes são compostos por objectos.

Se utilizarmos os diagramas de instalação na modelação de processos de negócios, os elementos de processamento são as unidades organizacionais e os trabalhadores enquanto que os componentes de software são os processos e documentos usados pelas unidades organizacionais e pelos trabalhadores.

Seguem-se dois exemplos de diagramas de instalação. O primeiro exemplo diz respeito a uma versão simplificada do serviço 118 da Portugal Telecom numa versão cliente/servidor. O segundo exemplo representa o equipamento hardware tipicamente existente numa configuração doméstica.

Exemplo 8.3: Diagrama de Instalação do serviço 118 da PT.

Considere-se uma versão simplificada do serviço 118 da Portugal Telecom numa sua versão cliente/servidor em que o cliente é uma aplicação previamente instalada e configurada num PC com MS-Windows. (Nota: existe também este serviço na versão Web em `http://net118.telecom.pt`).

Figura 8.7: Diagrama de instalação do Exemplo 8.3.

Pelo facto do diagrama de instalação apresentar componentes, todos os elementos apresentados têm de ser instâncias, neste caso são apresentadas instâncias de nós e de componentes. Outro aspecto relevante

deste exemplo é a representação, neste diagrama de instalação, da existência de vários PC através do caracter "*" colocado no canto superior direito do nó "PC".

Exemplo 8.4: Diagrama de Instalação do Sistema de Trabalho Doméstico.

A Figura 8.8 apresenta o diagrama de instalação correspondente a um sistema de trabalho doméstico constituído por um PC, com alguns equipamentos adicionais, por exemplo: uma impressora, um monitor, colunas de som, e um modem. O modem permite a ligação à Internet através de um determinado ISP (*Internet Service Provider*).

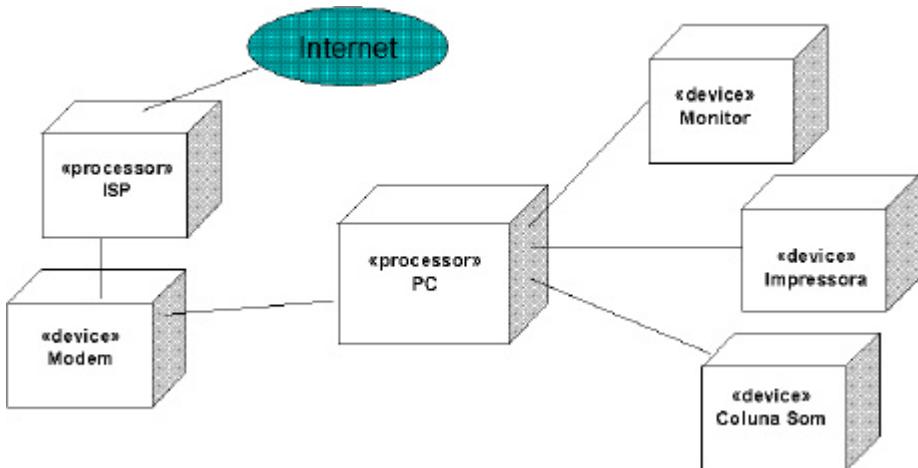
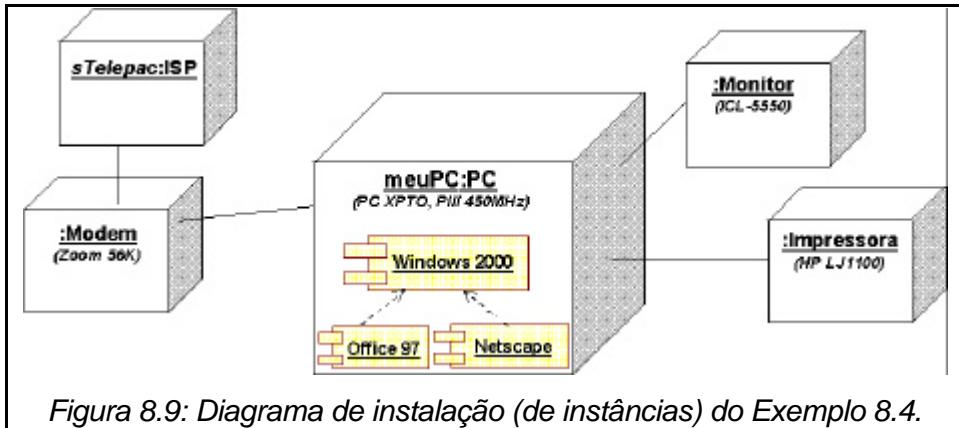



Figura 8.8: Diagrama de instalação (de tipos) do Exemplo 8.4.

Caso se pretendesse ilustrar uma configuração particular do diagrama apresentado e/ou ilustrar os componentes de software que deveriam existir numa determinada configuração, ter-se-ia de desenhar um diagrama de configuração de nível de instâncias. A Figura 8.9 ilustra uma possível configuração (instanciação) desenhada a partir do diagrama da Figura 8.8.

Figura 8.9: Diagrama de instalação (de instâncias) do Exemplo 8.4.

8.5 Exercícios

Ex.58. Pretende-se o diagrama de componentes correspondente à aplicação `ex-pipes` desenvolvido em linguagem C, com os seguintes módulos: `ex-pipes.c` `util.c` `server.c` `client.c`, e com dependências definidas pelo seguinte *makefile*:

```
CC = gcc
CFLAGS = -g
      ex-pipes : ex-pipes.o util.o server.o client.o
      $(CC) -g -o ex-pipes ex-pipes.o util.o server.o
client.o
```

Ex.59. Pretende-se o diagrama de componentes correspondente à página Web `http://www.tvi.pt/` com o seguinte conteúdo (tenha em consideração os componentes (ficheiros) representados a negrito.):

```
<html>
       <head>
             <meta http-equiv="content-type" content="text/html">
             <title>TVI OnLine</title>
       </head>
```

```
        <frameset rows="296,*" border="0" frameborder="NO"
framespacing="0">
                <frame src="index_hdr.html" name="hdr" noresize>
                <frame src="index_ix.html" name="ix" noresize
scrolling="NO">
        </frameset>
        <noframes>
                <bodbgcolor="#000000"
background="HmPG/directoIX_BG.jpg">
                </body>
        </noframes>
</html>
```

Ex.60. Pretende-se o diagrama de instalação da infra-estrutura computacional de apoio às suas aulas práticas. Considere apenas os nós existentes e os seus tipos de comunicação.

Ex.61. Alterar o diagrama produzido no exercício anterior de modo a incluir a descrição dos postos de trabalho e os componentes de software mais relevantes (e.g., servidor Web, ferramentas de trabalho do tipo Rose ou VisualStudio, servidor BD, sistema operativo).

Ex.62. Considere o serviço 118 da PT conforme introduzido no Exemplo 8.3. Modifique o exemplo dado tendo em consideração que o serviço é acedido através de um cliente/*browser* Web.

Ex.63. Pretende-se o diagrama de instalação para modelar a seguinte situação:

"Uma empresa industrial está estruturada em quatro departamentos: produção, comercial, controlo da qualidade, e administrativo-financeiro. Cada um destes departamentos tem um director respectivo. O director-geral é o responsável pela coordenação e supervisão de todos os departamentos. O departamento administrativo-financeiro está estruturado em duas secções, respectivamente a secção administrativa e a secção financeira."

Sugestões:

(1) Considere que os recursos do negócio (unidades orgânicas e as pessoas) são nós do diagrama a desenhar.

(2) Represente, através de estereótipos, o tipo das associações existentes entre nós.

Capítulo 9 - UML – ASPECTOS AVANÇADOS

Tópicos

- Introdução
- A Arquitectura do UML
- Mecanismos de Extensão
- Perfis UML
- Sistemas de Componentes e Reutilização
- Tipos Parametrizáveis
- XMI – XML Metadata Interchange
- Conclusão

9.1 Introdução

Vimos ao longo dos capítulos anteriores os principais aspectos da linguagem UML. Em particular, vimos a sua estrutura de conceitos e a sua aplicação segundo diferentes perspectivas, designadamente segundo a perspectiva de modelação de casos de utilização, estrutural, comportamental e arquitectural. Analisámos, nos capítulos anteriores, o UML segundo a perspectiva da sua utilização.

Neste capítulo introduzimos alguns aspectos adicionais, que consideramos avançados, que permitem ao utilizador uma compreensão mais profunda do UML e, por conseguinte, uma melhor aplicação. Nomeadamente, analisamos os seguintes aspectos:

- A estrutura e arquitectura do UML (i.e. como é que o próprio UML se encontra definido e organizado).

- Mecanismos de extensão do UML (i.e. como estender o UML de forma a contemplar exigências de modelação mais recentes e futuras). Serão apresentados alguns exemplos concretos de conjuntos de elementos que estendem o UML, os quais são designados por perfis.
- Noções gerais sobre o conceito de componente, sistemas de componentes (*toolkits* e *frameworks*), famílias de aplicações e reutilização.
- Como representar em UML tipos parametrizáveis, em particular classes parametrizáveis e colaborações parametrizáveis (no contexto de padrões de desenho).
- O standard OMG de interoperação de modelos UML em XML, o XMI. O objectivo do XMI é que os modelos UML possam ser exportados/importados e, por conseguinte, usados por diferentes ferramentas CASE de forma independente.

(Embora a leitura deste capítulo não seja essencial para uma utilização corrente do UML, tal leitura permite dar uma panorâmica mais abrangente e profunda da sua arquitectura e correspondentes capacidades.)

9.2 A Arquitectura do UML

9.2.1 A Estrutura do UML a Quatro Camadas

O UML está estruturado numa arquitectura de quatro camadas conforme sugerido na Figura 9.1: meta-metamodelo; metamodelo; modelo; e objectos do utilizador.

Figura 9.1: A arquitectura do UML a quatro camadas.

Este tipo de arquitectura a quatro camadas apresenta uma infra-estrutura adequada para a definição da semântica associada a modelos complexos. As camadas distinguem-se pelo nível de generalidade e de abstracção dos seus elementos constituintes.

Na camada mais baixa da arquitectura – "Camada Objectos do Utilizador" – encontram-se os elementos que constituem as instâncias concretas dos elementos representados num modelo definido pelo utilizador do UML (i.e., o analista ou consultor, e não o utilizador do sistema final): os objectos, os cenários de um caso de utilização, as instâncias de componentes, etc. Por exemplo, o objecto `Termo0033`, o valor `proposto`, a operação `muda_estado`, ou a instância do *servlet* `termoServlet0444` ilustram elementos desta camada.

Na segunda camada – "Camada Modelo" – encontram-se os elementos que constituem as abstracções definidas pelo utilizador do UML no seu processo de definição de modelos. Por exemplo, a classe `Termo`, o atributo `estado`, a operação `mudaEstado`, o componente `TermoServlet` ou o caso de utilização '`Propor Termo`" ilustram elementos desta camada.

Na terceira camada – "Camada Metamodelo" – encontram-se os elementos que constituem as abstracções definidas pelos criadores do UML, ou sejam, os elementos que constituem a estrutura de conceitos

do UML, com base nos quais se podem definir modelos (daí a designação de "metamodelo"). Por exemplo: `Classe`, `Atributo`, `Operação`, `Componente`, `Nó`.

Estas três primeiras camadas são aquelas a que a generalidade dos utilizadores UML acabam por recorrer na maior parte das vezes, quer como utilizadores (da terceira camada) quer como criadores (de elementos das duas primeiras camadas). Por isso é natural que se coloque a questão: para quê a quarta camada? De facto, esta camada – "Camada Meta-metamodelo" – será raramente usada pelo utilizador normal do UML. No entanto, é necessária para os responsáveis pelo desenho e comparação deste tipo linguagens. Esta camada permite definir os elementos que serão responsáveis pela especificação de metamodelos. Por exemplo: `MetaClasse`, `MetaAtributo`, `MetaOperação`, `MetaComponente`, `MetaNó`.

Sai fora dos objectivos deste livro aprofundar esta quarta camada da arquitectura do UML. Por outro lado, ilustra-se na secção seguinte uma panorâmica geral sobre a camada do metamodelo.

9.2.2 A Camada Metamodelo

O UML é descrito através de um modelo, designado "metamodelo" por ser um modelo que descreve outro modelo, também ele descrito em UML. O metamodelo UML encontra-se estruturado nos seguintes pacotes lógicos: `Foundation`, `Behavioral Elements` e `Model Management` (ver Figura 9.2), os quais por sua vez são decompostos noutros subpacotes.

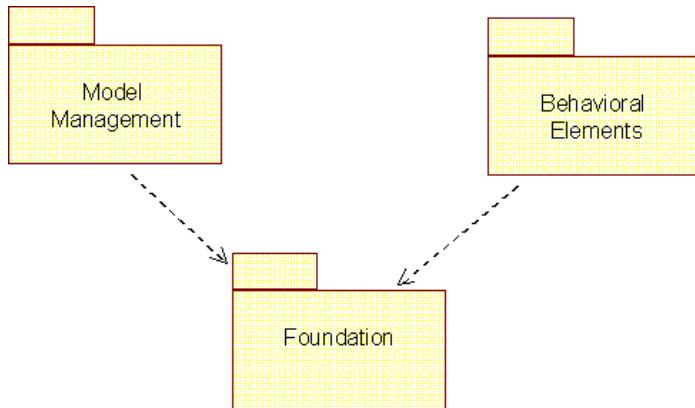



Figura 9.2: Os pacotes de alto nível do metamodelo UML.

O metamodelo é descrito de forma semi-formal usando as seguintes perspectivas: (1) sintaxe abstracta, (2) regras e (3) semântica. A sintaxe abstracta é providenciada através de um modelo descrito num subconjunto do próprio UML, consistindo nos diagramas de classes UML e por uma descrição em linguagem natural. As regras são especificadas através de linguagem natural e da linguagem formal OCL. Por último, a semântica é descrita principalmente em linguagem natural. De facto, o metamodelo UML é descrito por uma combinação de notações gráficas, linguagem natural e linguagem formal, o que segundo os seus proponentes oferece um compromisso adequado entre expressividade, rigor e facilidade de leitura [OMG99].

Pacote `Foundation`

O pacote `Foundation` é o componente da linguagem que especifica a estrutura estática dos modelos. Contem os seguintes pacotes `Core`, `Extension Mechanisms`, e `Data Types` (ver Figura 9.3).

O pacote `Core` especifica os conceitos básicos do metamodelo e define a estrutura arquitectural que permite a associação de construtores adicionais, tais como metaclasses, metaassociações, e metaatributos.

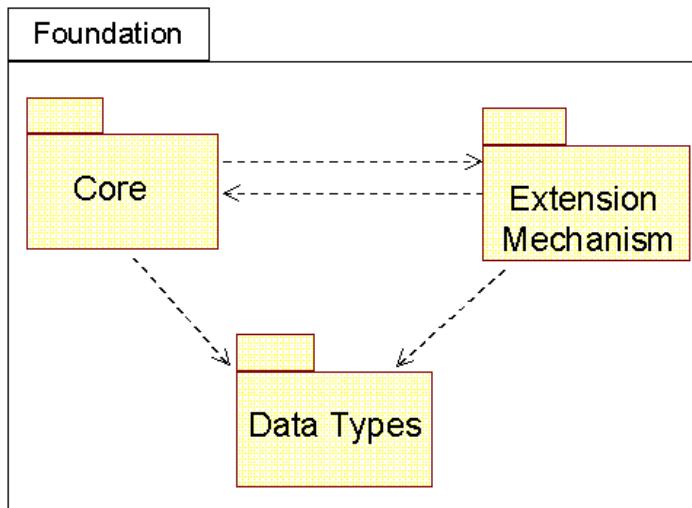
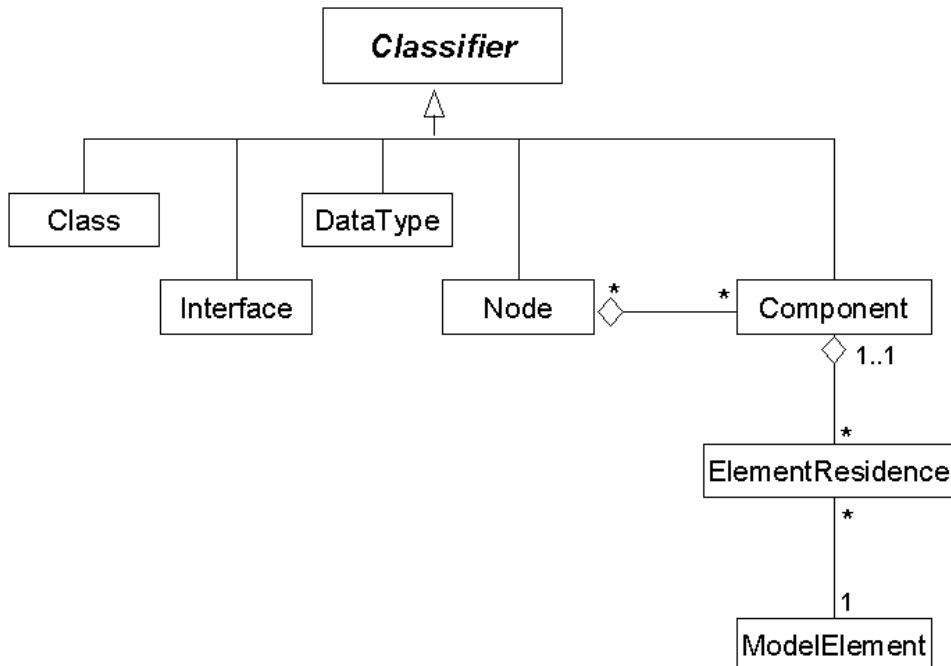



Figura 9.3: Os subpacotes do pacote `Foundation`*.*

O pacote `Core` define os principais construtores do metamodelo (abstractos e concretos) necessários ao desenvolvimento de modelos de objectos. Os construtores abstractos são aqueles que não podem ser usados directamente pelo utilizador e que apenas são concebidos para dar estrutura e organização ao metamodelo UML, tais como `ModelElement`, `GeneralizableElement` e `Classifier`. Por outro lado, os construtores concretos são aqueles que são usados (i.e., instanciáveis) directamente pelo utilizador, tais como `Class`, `Attribute`, `Operation` e `Association`.

Um **classificador** (***classifier***) é um supertipo definido no metamodelo UML que é usado extensivamente ao longo da especificação do UML sempre que se pretende referir um elemento que descreve estrutura e comportamento, ou seja, sempre que se pretende referir indistintamente a qualquer dos seguintes conceitos: *classe*, *interface*, *tipo de dados*, *caso de utilização*, *actor*, *sinal*, *subsistema*, *nó* ou *componente.*

A Figura 9.4 ilustra o diagrama de classes dos *classifiers* definidos no pacote `Core|Foundation`. (Note-se que a explicitação dos outros *classifiers*, por exemplo *actor* ou *caso de utilização* são definidos noutros pacotes.) Segundo uma descrição mais rigorosa, `Classifier` é uma super metaclasse (meta, porque é uma classe do metamodelo),

abstracta, que é especializada por outras metaclasses, tais como `Class`, `Interface` ou `Node`, em que estas são metaclasses concretas.

Figura 9.4: Os classifiers definidos no subpacote `Core|Foundation`.

O pacote `Extension Mechanisms` especifica como os elementos do modelo podem ser configurados e estendidos com nova semântica. Define a semântica de estereótipos, restrições e marcas com valores através da definição dos seguintes construtores concretos, respectivamente `Stereotype`, `Constraint` e `TaggedValue`.

Por fim, o pacote `Data Types` define as estruturas de dados básicas da linguagem, tais como tipos de dados primitivos (e.g., `Integer`, `String`, `Time`), enumerados (e.g., `Boolean`, `AggregationKind`, `VisibilityKind`), e outros (e.g., `Expression`, `Mapping`, `Name`, `Multiplicity`).

Pacote ***`Behavioral Elements`***

O pacote `Behavioral Elements` contem os subpacotes `Common Behavior`, `Collaboration`, `Use Cases`, `Activity Graphs`, e `State Machines` (ver Figura 9.5) que permitem representar o comportamento e a dinâmica de um sistema. Incluem os conceitos discutidos ao longo dos Capítulos 5 e 7, tal como os seus nomes sugerem.

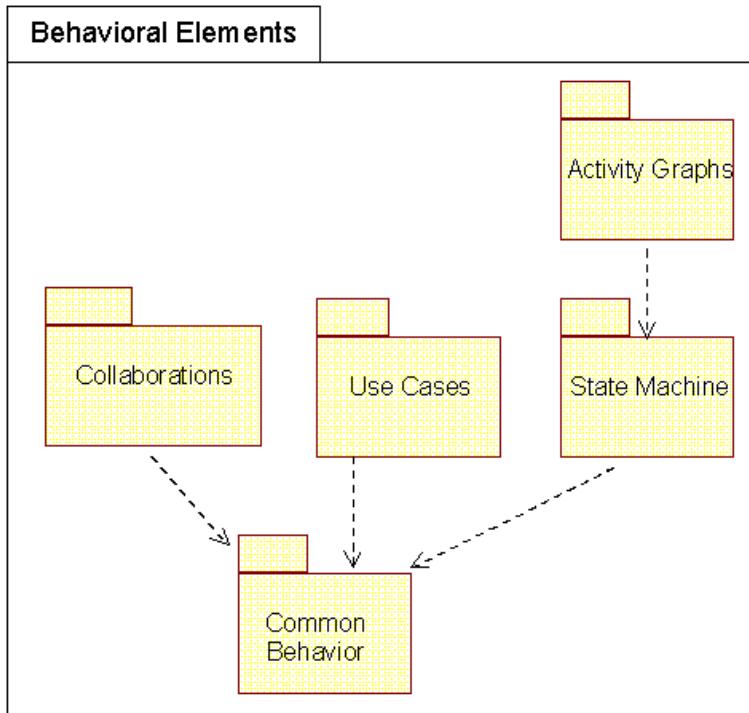



Figura 9.5: Os subpacotes do pacote `Behavioral Elements`*.*

Pacote ***`Model Management`***

O pacote `Model Management` depende do pacote `Foundation` e consiste num diagrama de classes que ilustra como os elementos `Model`, `Package` e `Subsystem` se organizam e relacionam entre si. Estes elementos servem principalmente como unidades de agrupamento e de organização dos outros elementos do modelo.

9.3 Mecanismos de Extensão

O UML providencia um número elevado de conceitos e notações particularmente concebidos de forma a satisfazer os requisitos típicos de modelação de software. Contudo, podem surgir situações em que se torna desejável a introdução de conceitos e/ou de notações adicionais para além dos definidos originalmente no momento da definição do standard. Para contemplar tais situações, conforme introduzido na Secção 4.6.2, o UML providencia diversos mecanismos que permitem estendê-lo de forma consistente: restrições, marcas e estereótipos. Estes mecanismos permitem [OMG99]:

- Introduzir novos elementos de modelação para uma maior expressividade e compreensão dos modelos UML a criar.
- Definir itens standard que não são considerados suficientemente interessantes ou complexos para serem definidos directamente como elementos do metamodelo UML.
- Definir extensões específicas das linguagens de implementação ou específicas dos processos de desenvolvimento.
- Associar arbitrariamente informação semântica e outra aos elementos do modelo.

Estes mecanismos aplicam-se aos elementos do modelo, não às suas instâncias. Representam, portanto, extensões à própria linguagem que permitem alterar a estrutura e semântica dos modelos criados.

A Figura 9.6 ilustra a sintaxe abstracta dos mecanismos de extensão do UML. Note-se a definição e relação entre as metaclasses `Stereotype`, `Constraint` e `TaggedValue`.

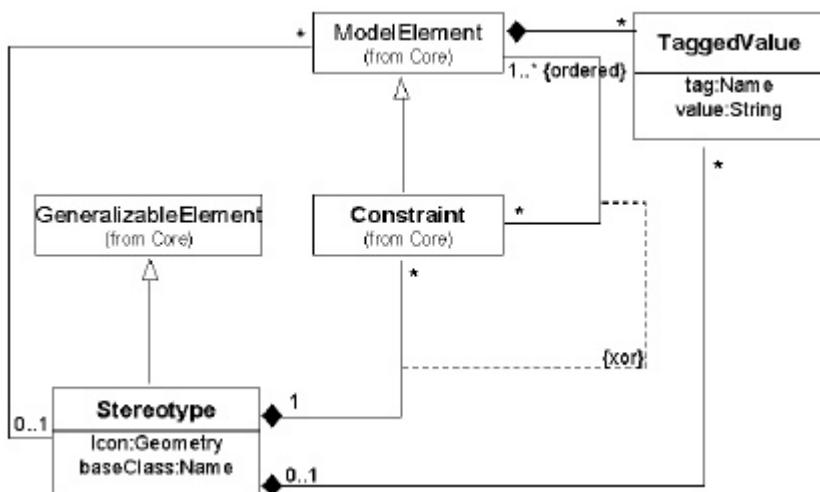



Figura 9.6: Mecanismos de extensão.

Restrições

Uma **restrição** consiste na especificação de semântica associada a um (ou a um conjunto ordenado de) elemento(s) do modelo. Tal especificação é escrita numa determinada linguagem de restrições, que pode ser mais ou menos formal. Por exemplo: OCL, linguagem de programação, notação matemática ou linguagem natural. Pelo facto da escolha da linguagem ser arbitrária, as restrições são um mecanismo de extensão [OMG99].

No metamodelo uma restrição (`Constraint`) deriva directamente de um `ModelElement` descreve as restrições semânticas que esse `ModelElement` tem de satisfazer. Por outro lado, restrições associadas a um `Stereotype` aplicam-se a cada `ModelElement` ligados ao `Stereotype`.

Marcas com Valor

Uma **marca com valor** é um par `<marca,valor>` que permite associar arbitrariamente informação a qualquer elemento do modelo. Uma marca é um nome arbitrário, havendo contudo alguns nomes predefinidos como elementos standard (e.g., *new*, *destroyed*, *transient*, *self*, *disjoint*, *xor*). O significado de cada marca é intencionalmente específico do

contexto da sua utilização, podendo ser determinado pelo seu utilizador ou por convenções das ferramentas de suporte.

Estereótipos

Um **estereótipo** providencia um mecanismo de classificação de elementos, tal que eles se comportem, de alguma forma, como se fossem instâncias de novos construtores definidos no metamodelo.

Um estereótipo pode especificar adicionalmente restrições e marcas com valor que serão aplicadas às suas instâncias. Um estereótipo pode ser usado para especificar diferenças de utilização ou de semântica entre dois elementos com uma estrutura relativamente semelhante.

Um estereótipo tem a indicação da sua classe base e, opcionalmente, uma representação gráfica (ícone) correspondente. A classe base de um estereótipo é uma classe no metamodelo UML (i.e., não é um elemento de modelação criado pelo utilizador) tal como `Class`, `Association` ou `Refinement`. Um estereótipo pode ser um subtipo de um ou mais estereótipos existentes, dos quais herda as restrições e marcas com valor. Adicionalmente, um estereótipo pode adicionar as suas próprias restrições, a sua própria lista de marcas com valor, e um novo ícone para representação visual.

Se um elemento do modelo (de nível utilizador) é classificado por um determinado estereótipo, a classe base desse elemento tem de coincidir com a classe base especificada para esse estereótipo. As restrições e marcas com valores são implicitamente associadas a esse elemento do modelo.

9.4 Perfis UML

Os mecanismos de extensão ilustrados na Secção 9.3 devem ser considerados, tal como discutido na Secção 4.6, apenas como último recurso. Pelas suas próprias características de especificidade, as extensões não são normalmente compreendidas, suportadas e aceites pela generalidade dos utilizadores UML. De forma a organizar e

promover uma evolução mais pacífica destas extensões, os promotores do UML propuseram o conceito de “perfil”.

Um **perfil UML** é um nome dado a um conjunto predefinido de estereótipos, marcas com valor, restrições e ícones que conjuntamente especializam e configuram o UML para um determinado domínio de aplicação ou um determinado processo de desenvolvimento. Note-se que um perfil não estende o UML através da introdução de novos conceitos de base.

Na especificação 1.3 do UML são descritos dois perfis, designados como “perfis standard”: (1) perfil para processos de desenvolvimento de software; e (2) perfil para modelação de processos e estruturas de negócios.

Apresentamos de seguida, a título de exemplo, algumas considerações sobre os dois perfis referidos, bem como o perfil de modelação de aplicações Web baseado no trabalho de Jim Conollen [Conallen00].

Existe um razoável conjunto de outros esforços de extensão do UML para vários domínios de aplicação (note-se que nem todos esses esforços serão reconhecidos pela OMG como perfis UML). Referem-se a título de exemplo: (1) as extensões de Eriksson-Penker para modelação de negócio (*Eriksson-Penker Business Extensions*) [Penker00] suportadas por exemplo pela ferramenta Qualiware; (2) as extensões para modelação de negócio providenciadas na ferramenta ProVision Workbench, da empresa Proforma [Proforma00]; (3) extensões para modelação de dados, propostas pela Rational [Rational00]; ou (4) extensões para modelação de aplicações baseadas em agentes de software [Odell00].

(Nota: Sai do âmbito deste livro aprofundar a descrição dos perfis UML, tal como detalhar a sua aplicação em situações concretas.)

9.4.1 Perfil para Processos de Desenvolvimento de Software

O perfil “Processos de Desenvolvimento de Software” consiste na definição de um conjunto de estereótipos (actualmente não foram definidos quaisquer marcas com valor ou restrições) usados tipicamente

em processos de desenvolvimento de software inspirados no *Unified Process* [Jacobson99], em particular pelo RUP ou o ICONIX (ver Capítulos 10 a 11).

Classe do Metamodelo	**Nome do Estereótipo**
Model	*use-case model*
Model	*analysis model*
Model	*design model*
Model	*implementation model*
Package	*use-case system*
Package	*analysis system*
Subsystem	*design system*
Subsystem	*implementation system*
Package	*analysis package*
Subsystem	*design subsystem*
Subsystem	*implementation subsystem*
Package	*use-case package*
Package	*analysis service package*
Subsystem	*design service subsystem*
Class	*boundary*
Class	*entity*
Class	*control*
Association	*communicate*
Association	*subscribe*
Collaboration	*use-case realization*

Tabela 9.1: Resumo dos estereótipos definidos no perfil "Processos de Desenvolvimento de Software".

Estereótipos de Modelos, Pacotes e Subsistemas

Um sistema modelado com base no *Unified Process* consiste em vários modelos relacionados. Esses modelos são caracterizados pela fase do ciclo de vida que representam, designadamente: casos de utilização; análise; desenho; e implementação.

Casos de Utilização

O modelo de casos de utilização («use case model») especifica os serviços que um sistema providencia do ponto de vista dos seus utilizadores.

Um sistema de casos de utilização («use case system») é um pacote de nível inicial ou de topo, e contém outros pacotes (de estereótipo «use case package») e/ou casos de utilização e relações respectivas. Por fim, um «use case package» é um pacote que contém apenas casos de utilização e relações.

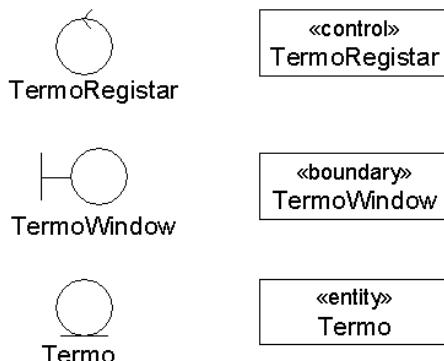
Análise

Um modelo de análise («analysis model») é um modelo cujo pacote de mais alto nível é do estereótipo «analysis system». Este pacote contém pacotes de análise (de estereótipo «analysis package») e/ou pacotes de serviços (de estereótipo «analysis service package») e/ou classes de análise (de estereótipos «entity», «boundary» e «control») e respectivas relações. Um pacote de análise contém pacotes de análise, pacotes de serviços de análise, classes de análise e relações. Por fim, um pacote de serviço de análise contém apenas classes de análise e relações.

Desenho

Um modelo de desenho («design model») é um modelo cujo subsistema de mais alto nível é do estereótipo «design system». Um sistema de desenho contém subsistemas de desenho (de estereótipo «design subsystem») e/ou subsistemas de serviços de desenho (de estereótipo «design service subsystem») e/ou classes de desenho e respectivas relações. Um subsistema de desenho contém outros subsistemas de desenho, subsistemas de serviços de desenho, classes de desenho e relações. Por fim, um subsistema de serviço de desenho contém apenas classes de desenho e relações.

Implementação

Um modelo de implementação («implementation model») é um modelo cujo subsistema de mais alto nível é do estereótipo «implementation system». Um sistema de implementação contém subsistemas de implementação (de estereótipo «implementation subsystem») e/ou componentes e respectivas relações. Um subsistema de implementação comtém outros subsistemas de implementação, componentes e relações.

Estereótipos de Classe

Definem-se três estereótipos para as classes de análise: «entity», «boundary» e «control» (ver Figura 9.7). No Capítulo 11 (Secção 11.5.3) é efectuada uma discussão mais detalhada da aplicação destes estereótipos.

Para as classes de desenho não são definidos quaisquer estereótipos particulares.

Entidade («entity»)

Uma entidade («entity») é uma classe passiva, no sentido que os seus objectos não iniciam interacções por si próprios. Este tipo de objectos participa em diferentes realizações de casos de utilização e geralmente persiste para além das interacções em que participa (i.e., são objectos persistentes).

Figura 9.7: Exemplos de estereótipos de classes do perfil de Processos de Desenvolvimento de Software.

Controlo («control»)

Um controlo («control») é uma classe cujos objectos controlam as interacções entre colecções de objectos. Geralmente este tipo de classes tem um comportamento específico para determinado caso de utilização e normalmente as suas instâncias não persistem para além das interacções em que participam (i.e., são objectos não persistentes).

Fronteira («boundary»)

Uma fronteira ou interface («boundary») é uma classe que se encontra na fronteira de um sistema, contudo, ainda dentro dele. A responsabilidade destes objectos é mediar a comunicação entre os actores externos do sistema e os outros objectos internos ao sistema.

Estereótipos de Associação

São apenas definidos neste perfil UML dois estereótipos particulares para associações entre classes: «communicate» e «subscribe».

Comunicação («communicate»)

A comunicação («communicate») é uma associação entre actores e casos de utilização, que traduz o envio de mensagens do actor para o caso de utilização e/ou vice-versa. Pode também ser usada entre as classes de interface, controlo e entidade, e entre actores e classes de interface. Se for relevante, pode-se explicitar a direcção da comunicação através de um adorno de navegabilidade (por omissão, assume-se comunicação bidireccional).

Subscrição («subscribe»)

A subscrição («subscribe») é uma associação entre duas classes-estado, cujos objectos da classe subscritora são notificados quando um determinado evento ocorre nos objectos da classe destino (designada por classe “publicadora”). A associação inclui a especificação do conjunto de eventos, cuja ocorrência implica que o subscritor seja notificado.

9.4.2 Perfil para Modelação de Negócios

O perfil "Modelação de Negócios" consiste na definição de um conjunto de estereótipos (não foram definidos quaisquer marcas com valor ou restrições) e ilustra as capacidades do UML para modelar não apenas sistemas de software, mas outras realidades, neste caso "o mundo das organizações".

A Tabela 9.2 resume os principais estereótipos definidos neste perfil UML, destacando-se entre outros, os conceitos de unidade da organização («organization unit»), unidade trabalho («work unit»), trabalhador («worker») e entidade («entity»).

Classe do Metamodelo	**Nome do Estereótipo**
Model	*use-case model*
Package	*use-case system*
Package	*use-case package*
Model	*object model*
Subsystem	*object system*
Subsystem	*organization unit*
Subsystem	*work unit*
Class	*worker*
Class	*case worker*
Class	*internal worker*
Class	*entity*
Collaboration	*use-case realization*
Association	*communicate*
Association	*Subscribe*

Tabela 9.2: Resumo dos estereótipos definidos no perfil "Modelação de Negócios".

No contexto deste perfil, um trabalhador («worker») actua dentro de um determinado negócio, participa em casos de utilização e interactua com

outros trabalhadores e entidades («entity»). Os trabalhadores internos («internal worker») interagem com outros trabalhadores dentro do negócio, enquanto que os trabalhadores de interface («*case worker*») interagem com os actores externos ao sistema.

Os objectos interagem segundo dois tipos de associações: de comunicação («communicate») e de subscrição («subscribe»), segundo o modelo de subscrição/publicação.

Um conjunto de entidades agregadas de forma combinada constitui uma unidade de trabalho («work unit»). Por fim, um conjunto de unidades de trabalho, de classes e de associações agregadas de forma combinada constitui uma unidade organizacional («organization unit») de suporte ao negócio.

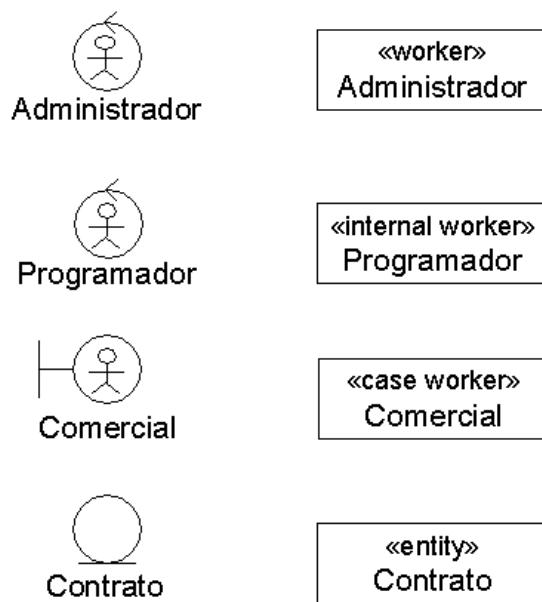



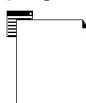
Figura 9.8: Exemplos de estereótipos de classes do perfil de modelação de negócios.

A Figura 9.8 ilustra a representação gráfica de alguns dos estereótipos definidos no perfil de modelação de negócios.

9.4.3 Perfil para Modelação de Aplicações Web

Jim Conallen, no seu livro "*Building Web Applications with UML*" [Conallen00], propõe um processo de modelação de aplicações Web baseado em UML.

Aplicações Web são aplicações baseadas em diversos modelos computacionais (e.g., computação no cliente, computação no servidor, computação distribuída entre cliente e servidor) e tecnologias (e.g., HTTP, HTML, *frames*, âncoras, linguagens de *scripting*, DOM, XML, Java/Beans, ActiveX/COM, objectos distribuídos) que estão na base da generalidade dos actuais sistemas de informação das organizações em todo o mundo.

Conollen propõe um número significativo de extensões ao UML, designado no seu conjunto como "WAE" (*Web Application Extension*), designadamente os seguintes estereótipos, derivados a partir de:

- Metaclasses Class: Server Page, Client Page, Form, Frameset, Target, JavaScript Object, ClientScript Object.
- Metaclasses Association: Link, Targeted Link, Frame Content, Submit. Builds, Redirect, IIOP, RMI.
- Metaclasses Attribute: Input Element, Select Element, Text Area Element.
- Metaclasses Component: Web Page, ASP Page, JSP Page, Servlet, Script Library.

O processo de desenvolvimento deste tipo de aplicações é clássico nas fases de especificação de requisitos e de análise, mas é inovador na fase de desenho. É nesta fase que é aplicado o WAE de forma adequada. Apresenta-se de seguida, a título de exemplo, a definição do estereótipo `Client Page` e do estereótipo `Link` [Conallen00].

***Client Page* (Página Cliente)**

Classe base	`Class`
Descrição	Uma instância de uma «client page» é uma página Web formatada em HTML, sendo uma combinação entre dados, apresentação e mesmo lógica. Estes elementos são apresentados pelos *browsers* Web e podem conter *scripts* que serão

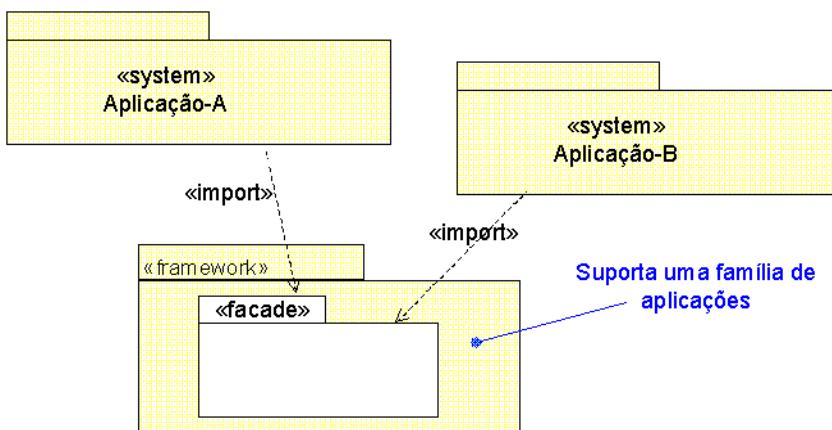
	interpretados pelos mesmos. As funções destes elementos correspondem às funções definidas nas suas tags de *script*. Os atributos destes elementos correspondem às variáveis declaradas nas suas *tags* de *script*, acessíveis por qualquer função da página (*page scope*). As páginas cliente podem ter associações com outras páginas cliente ou páginas servidor.
Ícone	
Restrições	Nenhuma
Marcas com valor	`TitleTag`: O título da página. `BaseTag`: O URL base para resolução de endereços de URL relativos. `BodyTag`: O conjunto de atributos definidos na *tag* <body>, os quais definem os atributos do texto por omissão e de cores.

***Link* (Ligação)**

Classe base	`Association`
Descrição	Uma ligação (*link*) é uma referência de uma página cliente para outra página («page»). Num diagrama de classes, uma ligação é uma associação entre uma «client page» e outra «client page» ou «server page». Uma ligação corresponde directamente à *tag* âncora (i.e., <a>) do HTML.
Ícone	Nenhum
Restrições	Nenhuma
Marcas com valor	`Parameters`: Uma lista de nomes de parâmetros que deverão ser no pedido para a página ligada.

Note-se, como anteriormente referido, que a definição de um estereótipo envolve a especificação da sua classe base (definido no metamodelo), uma descrição informal, e eventualmente de um ícone, e de restrições e marcas com valor correspondentes.

9.5 Sistemas de Componentes e Reutilização

9.5.1 Definição de Componente

A noção geral de **componente** corresponde a uma entidade que possa ser reutilizada. Assim um componente não se limita apenas e necessariamente à visão arquitectural ao nível de implementação (e.g., segundo as especificações Java Beans, Active-X, ou CORBA) por que é geralmente designado e reconhecido, mas também, por exemplo, por

- um conjunto de classes C++ (nível implementação)
- um conjunto de casos de utilização (nível de especificação de requisitos)
- um conjunto de diagramas de classes (nível de análise ou de desenho)

desde que tenham sido definidos de forma a serem reutilizáveis.

9.5.2 Famílias de Aplicações

Devem ser reconhecidas **famílias de aplicações** e definidas arquitecturas de sistemas de componentes especificamente desenhadas e implementadas para suportar essas respectivas famílias.

Figura 9.9: Um framework suportando o desenvolvimento de uma família de aplicações.

A Figura 9.9 ilustra um exemplo de uma aplicação (de estereótipo «system») que (re)utiliza os componentes de um determinado sistema de componentes (de estereótipo «framework») através de um pacote de interface, conhecido genericamente por "fachada" (de estereótipo «facade»).

Uma **fachada** permite esconder os detalhes de concepção e implementação de um determinado sistema de componentes, oferecendo uma interface mais coerente e mais simples de usar (importar) pelas aplicações específicas.

9.5.3 Sistemas de Componentes

Distinguem-se dois tipos principais de sistemas de componentes: *toolkits* e *frameworks*.

Um ***toolkit*** (biblioteca de componentes) consiste num conjunto bem definido, integrado e reutilizável de tipos que providenciam funcionalidades de propósitos gerais. Um conjunto de classes reutilizáveis que permite, por exemplo, a gestão de listas, pilhas, tabelas de *hash* é um *toolkit*. O JGL, *Generic Collection Library for Java* (ver mais informação em *www.objectspace.com*), é um exemplo de um *toolkit* para o sistema Java ao providenciar um conjunto de tipos reutilizáveis tais como: contentores (e.g., *queue, map, set, array*); algoritmos genéricos (e.g., ordenação, filtro); objectos-funções; e iteradores. As bibliotecas de I/O ou STL (*Standard Template Library*) [Plauger96] para C++ são outros exemplos. Os *toolkits* não impõem um desenho particular das aplicações; eles apenas providenciam funcionalidades genéricas que as aplicações podem utilizar, sendo portanto, que o seu foco consiste na reutilização de código. O *toolkit* é o equivalente OO às bibliotecas de sub-rotinas da programação estruturada.

Um ***framework* (infra-estrutura) software** é um conjunto de classes cooperantes que providencia funcionalidades comuns e reutilizáveis para um determinada família de (sistemas de) aplicações. Uma infra-estrutura providencia mecanismos para criação de aplicações particulares a partir da especialização de aplicações reutilizáveis e semi-completas. Exemplos de infra-estruturas são, entre outras:

- MFC (Microsoft Foundation Classes) da Microsoft, para desenvolvimento de aplicações para MS-Windows.
- Infra-estruturas de *middleware*, tais como o RMI da Sun, o DCOM da Microsoft, OrbixWeb da Iona, ou a VisiBroker da Inprise.
- Infra-estruturas aplicacionais, tais como os sistemas de gestão empresarial conhecidos como ERP (*entreprise resource planning*) de empresas como a SAP, Oracle, ou PeopleSoft.

Uma infra-estrutura de software define a arquitectura de um aplicação, ou seja, define a estrutura e organização global da aplicação, define como as classes e objectos podem colaborar, e define os próprios fluxos de execução possíveis. Uma infra-estrutura predefine os principais parâmetros do desenho de uma aplicação, deixando apenas para o desenhador/programador de uma aplicação as tarefas exclusivamente específicas da aplicação. Desta forma a ênfase de uma infra-estrutura consiste na reutilização do desenho (de famílias de aplicações).

A reutilização nestes sistemas apresenta tipicamente uma inversão de controlo entre a aplicação específica e a infra-estrutura sobre a qual aquela se baseia. Numa aplicação normal (i.e., sem ser suportada por uma infra-estrutura) o programador define explicitamente o corpo do programa principal, invocando a partir deste as rotinas, ou métodos de objectos, que pretende (re)usar. Por outro lado, nas aplicações suportadas em infra-estruturas, o programador reutiliza o corpo principal (i.e., a estrutura e comportamento), previamente definido, apenas desemvolvendo o código específico que o outro invoca transparentemente.

Os tipos exportados ou visíveis por uma infra-estrutura encontram-se organizados em um ou mais pacotes, ou interfaces, designados por "fachadas". Alguns desses tipos correspondem a classes abstractas que deverão ser redefinidas e/ou implementadas numa determinada (instância de) aplicação, enquanto que outros tipos correspondem a classes concretas, com métodos finais, associadas a operações predefinidas, mas também com métodos que deverão/poderão ser redefinidos de forma que se possa adaptar os processos de negócio a cada aplicação em particular. Estes últimos métodos são desencadeados/invocados em determinado ponto da execução da aplicação

transparentemente ao programador da aplicação, e são geralmente conhecidos por métodos invertidos (*callbacks*).

As aplicações concebidas e desenvolvidas com base em infra-estruturas são mais rápidas de desenvolver – pois o programador apenas tem de definir o código dos métodos invertidos necessários e/ou especializar classes abstractas –; e apresentam estruturas semelhantes entre aplicações da mesma família, o que permite (1) uma mais fácil manutenção; e (2) uma mais fácil utilização por parte dos seus utilizadores. Por outro lado, o programador perde a visão do "todo", perde alguma liberdade criativa já que um número significativo de decisões de desenho já foram previamente realizadas.

9.5.4 Reutilização

A reutilização no processo de desenvolvimento de software deve ser um objectivo em qualquer das fases envolvidas, começando desde logo, pela especificação de requisitos, até à fase de testes, passando pela fase de desenvolvimento, desenho e análise.

Introduz-se de seguida algumas noções gerais, mas fundamentais para uma adopção de uma aproximação de reutilização no processo de desenvolvimento de software conforme sugerido em [Jacobson97].

Especialização de Componentes

Conceito

De modo que um sistema de componentes possa ser efectivamente reutilizável é necessário que várias das suas componentes exportadas apresentem um determinado grau de **variabilidade**, i.e., que possam ser de alguma forma extensíveis. Diferentes tipos de mecanismos de variabilidade providenciam distintos modos de especialização de componentes.

Uma componente importada pode ser abstracta ou concreta. Componentes concretas podem ser reutilizáveis directamente sem qualquer alteração. Basicamente uma **componente concreta** para ser reutilizada basta que seja importada bem como a todas as componentes que ela utilize. **Componentes abstractas** são de alguma forma componentes incompletas ou genéricas, que têm de ser especializadas

antes de poderem ser utilizadas (e.g., supertipos, classes abstractas, *templates* com parâmetros).

Designa-se por **pontos de variação** de uma componente às possibilidades de variabilidade dessa componente. Uma **variante** é uma concretização dessa possibilidade. Uma componente abstracta é especializada através da afectação de uma ou mais variantes aos seus pontos de variação. Essa afectação consiste na integração da determinada variante na componente envolvida, cuja semântica depende do mecanismo de variabilidade específico. O conceito de "ponto de extensão", referido no Capítulo 5 a propósito da relação de extensão entre casos de utilização, é um possível exemplo do conceito de "ponto de variação".

Mecanismos de Variabilidade

Componentes abstractas são especificadas através de mecanismos de variabilidade, de entre os quais se destacam:

- Herança: Permite a criação de tipos por especialização de outros tipos (mais) abstractos nos seus pontos de variação. Um método virtual pode ser entendido como um ponto de variação que pode ser especializado através de herança.
- Parametrização: É o mecanismo típico usado para concretização de *templates* e de macros.
- Linguagens de configuração: Este mecanismo é usado para, de forma declarativa ou procedimental, se definirem relações entre componentes e respectivas variantes em configurações completas ou finais.
- Geração automática: Mecanismo de construção de componentes e suas relações a partir de representações de alto nível definidas em *templates* ou linguagens específicas (e.g., *wizards*, *builders*, geradores de ecrãs a partir de *scripts*).
- Extensões e pontos de extensão: Mecanismo usado em componentes de casos de utilização e em componentes de objectos, que consiste na associação de variantes (extensões) a pontos de variação (pontos de extensão).

Note-se que estes mecanismos de variabilidade não são exclusivos entre si. Pelo contrário, é normal a utilização de vários mecanismos em conjunto. Em [Jacobson97] é proposta com detalhe uma abordagem de reutilização de software, ao longo das diferentes fases do seu ciclo de vida, baseada principalmente nos mecanismos de variabilidade aqui referidos, que agregam de facto a generalidade dos mecanismos conhecidos.

9.6 Tipos Parametrizáveis

A noção de tipos parametrizáveis surgiu nalgumas linguagens de programação como um mecanismo complexo para reutilização de código. Por exemplo, esta noção é designada (e concretizada) em C++, por *templates* [Ellis90]; em Ada, por *generics* [DoD80]; ou em ML, por funções e tipos de dados polimórficos [Ullman94].

A noção de **tipo parametrizável** é usada para permitir que um conceito geral seja instanciado por um ou mais tipos específicos. Por exemplo, pode-se definir um pacote ou uma classe para implementar um tipo de lista genérica, e instanciá-lo depois de forma a produzir-se "listas de inteiros", "listas de imagens", "listas de registos de base de dados", etc. Este mecanismo é particularmente útil no desenho e implementação de *toolkits* e *frameworks* reutilizáveis. Por exemplo, a biblioteca STL (*Standard Template Library*) [Plauger96] usa *templates* C++ extensivamente.

O UML permite o desenho de tipos parametrizáveis. Em particular, qualquer *classifier* pode ser parametrizável, ou seja, pode-se ter classes, interfaces, componentes, nós (...) parametrizáveis. Pode-se ainda parametrizar um grupo de *classifiers* que colaborarem entre si. Apresentamos nesta secção, a título de exemplo, classes parametrizáveis e colaborações parametrizáveis (normalmente conhecidas por padrões de desenho).

9.6.1 Classes Parametrizáveis

A Figura 9.10 ilustra a representação UML de uma classe `Classe` parametrizável com os parâmetros `T1`, `T2`, `Tn` não definidos. Ilustra

ainda um esboço da classe `Vector` com o correspondente código em C++.

Podem-se conceber inúmeras aplicações de classes parametrizáveis. Uma aplicação típica é a definição de estruturas de dados genéricas, do tipo contentores, tais como listas, vectores, pilhas, ou *hashtables*; outra aplicação é a dos algoritmos genéricos do tipo iteradores, algoritmos de cópia, contagem, procura, filtro, ordenação, etc.

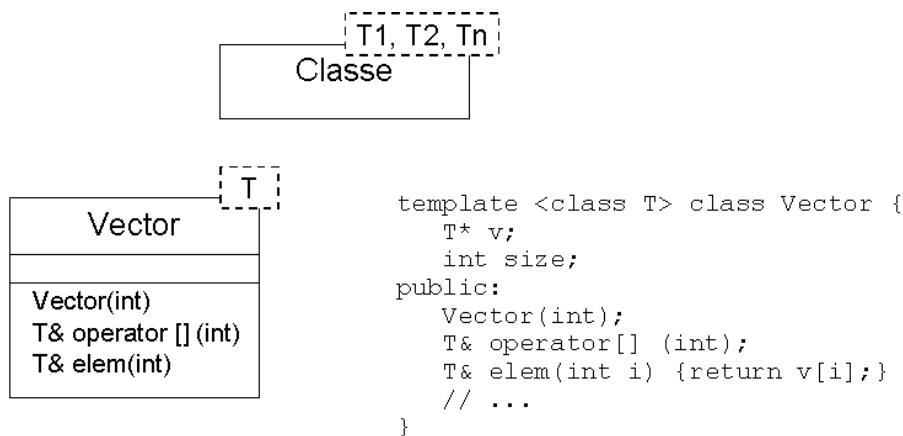



Figura 9.10: Representação UML de uma classe parametrizável.

A Figura 9.11 ilustra a classe parametrizável `Vector` com o parâmetro formal `T`. Para cada classe `C`, a classe `Vector<C>` é uma classe de vectores de objectos do tipo `C`, determinada pela instanciação da classe `Vector` com a classe actual `C` no lugar do parâmetro formal `T`. Essa instanciação é representada em UML por uma relação (de dependência) de estereótipo «bind».

As classes `VectorInteger` e `VectorImage` são criadas a partir da classe `Vector` pela substituição do parâmetro formal `T` por, respectivamente, os tipos `Integer` e `Image`. Notem-se nas duas formas equivalentes de se definirem classes a partir de uma classe parametrizável. Por exemplo, as classes `VectorInteger` e `Vector<Integer>` são equivalentes, sendo a primeira definida por ligação explícita e a segunda por ligação implícita.

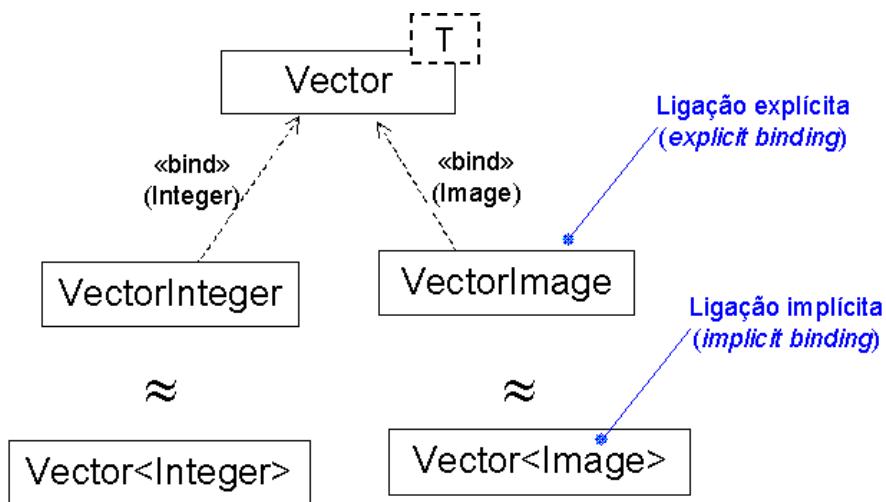
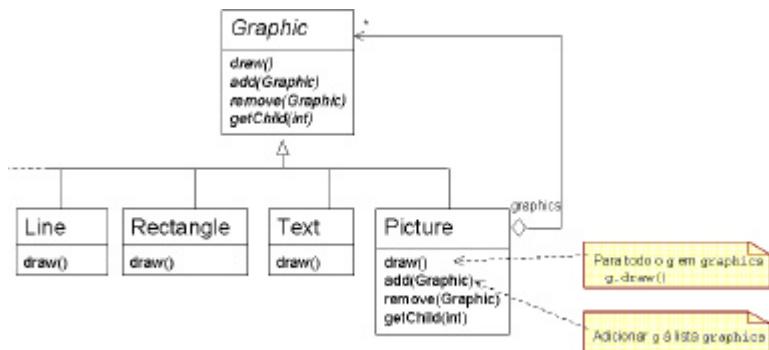



Figura 9.11: Exemplo da classe parametrizável `Vector<T>`*.*

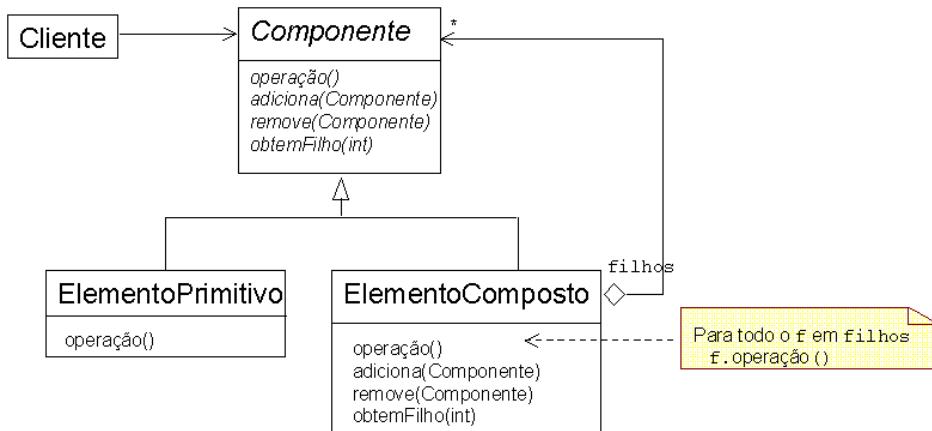
9.6.2 Padrões de Desenho

Um **padrão** descreve um problema que ocorre inúmeras vezes em determinado contexto, e descreve ainda a solução para esse problema, de modo que essa solução possa ser utilizada sistematicamente em distintas situações [Alexander77]. Embora Alexander se referisse a padrões no âmbito da engenharia civil, esse conceito foi adoptado para a engenharia informática, particularmente em padrões baseados em objectos e interfaces.

Um **padrão de desenho** é a adaptação do conceito de "padrão", proposto por Alexander, particularmente usado na fase de desenho segundo as abordagens orientadas por objectos. Ou seja, em que os padrões baseiam-se em objectos e interfaces. Um padrão de desenho pode ser especificado através de quatro elementos essenciais [Gamma-94]: (1) um nome, que permite facilmente a sua identificação e referência; (2) a descrição do problema a que o padrão deverá ser aplicado; (3) a solução, que consiste no desenho dos elementos constituintes e respectivas relações, responsabilidades, e colaborações; e (4) as consequências da aplicação do padrão, designadamente a identificação das vantagens e desvantagens envolvidas.

Os padrões de desenho aplicados ao desenvolvimento de software apresentam inúmeras vantagens, de entre as quais se destaca a identificação de soluções bem definidas para classes de problemas semelhantes, assim como, o facto de permitir a definição de construtores de desenho com um nível semântico elevado e facilmente reconhecidos por uma comunidade alargada de utilizadores.

De forma a esclarecer sucintamente o que são padrões de desenho e como são representados em UML introduz-se o padrão de desenho COMPOSTO (*Composite*), descrito no célebre livro do GoF (*Gang of Four*) [Gamma94].

O padrão de desenho COMPOSTO é usado para providenciar um acesso uniforme a estruturas de objectos compostas, permitindo que os seus clientes tratem da mesma forma quer objectos individuais quer objectos compostos.

Para motivação deste padrão, considere-se a construção de aplicações gráficas (e.g., editores gráficos) em que os utilizadores podem construir desenhos complexos a partir de componentes relativamente simples. Um utilizador pode agrupar componentes em componentes maiores, e estes, por sua vez, podem ser agrupados para constituir componentes ainda maiores. Uma aproximação simples para implementar este tipo de situações poderia consistir na definição de classes para primitivas gráficas do tipo `Texto` ou `Linha`, e de outras classes que funcionariam como contentores dessas classes primitivas. Contudo, esta aproximação apresenta o seguinte problema: o código que manipula essas classes tem de tratar de forma diferente elementos primitivos e elementos contentores, mesmo que os seus utilizadores os tratem de forma indistinta. A distinção entre estes dois tipos de elementos torna a aplicação mais complexa e difícil de manter. O padrão COMPOSTO descreve como usar a composição de forma recursiva de modo que os (objectos) clientes não tenham que fazer qualquer distinção entre elementos primitivos e compostos.

A chave do padrão COMPOSTO é uma classe abstracta que representa quer elementos primitivos quer elementos compostos. A Figura 9.12 ilustra o diagrama de classes correspondendo à aplicação deste padrão

ao exemplo sugerido, em que a classe abstracta é `Graphic` com operações comuns a todos os elementos gráficos (e.g., `draw()`) bem como operações comuns a todos os elementos compostos, tais como operações para acesso e gestão aos seus filhos (e.g., `add()`, `remove()`, `getChild()`).

Figura 9.12: Aplicação do padrão COMPOSTO no contexto de editores gráficos.

A Figura 9.13 apresenta o diagrama de classes correspondente à estrutura (genérica) do padrão COMPOSTO.

Figura 9.13: Estrutura genérica do padrão COMPOSTO.

Note-se que o padrão pressupõe uma colaboração definida entre os seguintes tipos de objectos (designados como "participantes" da colaboração):

- `Componente`: declara a interface para os objectos primitivos e compostos, seus descendentes; pode eventualmente implementar comportamento por omissão para algumas das operações (e.g., `Graphic`).
- `ElementoPrimitivo`: representa os elementos primitivos da composição; define o comportamento para os elementos primitivos (e.g., `Text`, `Line`, `Rectangule`).
- `ElementoComposto`: representa os elementos compostos; define o comportamento para os elementos compostos (e.g., `Picture`).
- `Cliente`: acede e manipula a composição de objectos através da interface de `Componente`.

Uma colaboração é representada em UML conforme ilustrado nas Figura 9.14 e 9.15, em que o nome da colaboração é apresentado numa oval com linha a tracejado, e são evidenciados os seus participantes através de relações de dependência, devidamente qualificadas com (o nome do) papel desempenhado.

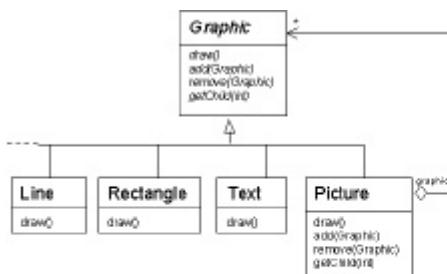
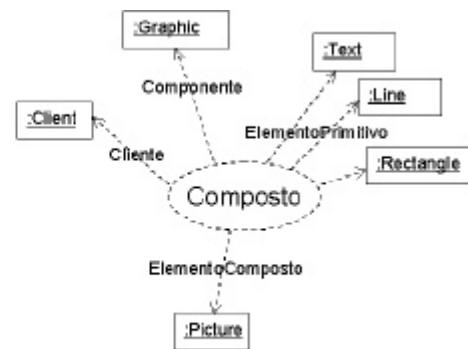



Figura 9.14: Diagrama de classes e colaboração relativa ao contexto de editores gráficos.

A Figura 9.15 ilustra o real interesse dos padrões de desenho por permitir aplicar a mesma solução, a problemas similares que se manifestam em diferentes contextos ou domínios de aplicação.

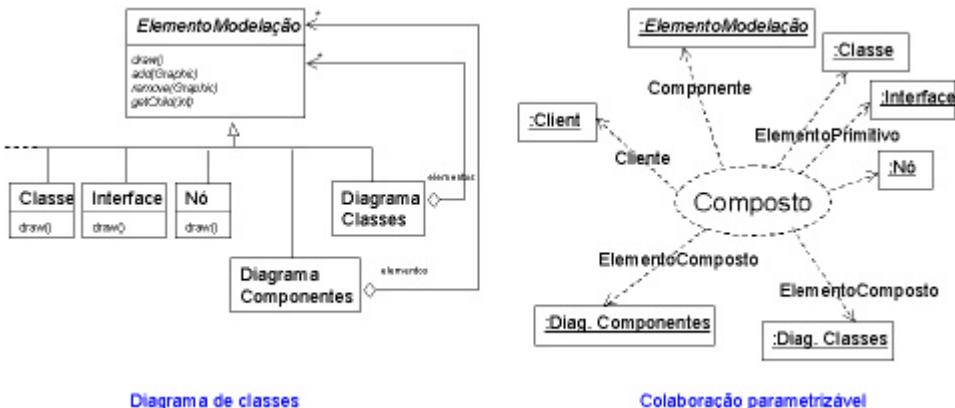



Figura 9.15: Diagrama de classes e colaboração relativa ao contexto de ferramentas CASE.

Neste caso, consideramos uma aplicação (e.g., qualquer das ferramentas CASE que são abordadas na Parte 4 do livro) que manipula extensivamente elementos de modelação, alguns dos quais primitivos (e.g., classes, interfaces, nós) e outros compostos (e.g., diagramas de classes, diagramas de componentes). Atente-se que o problema é, salvaguardando os detalhes específicos e o contexto de aplicação, semelhante ao descrito anteriormente, pelo que tem sentido a aplicação do padrão COMPOSTO. O aspecto crucial, é a identificação da classe abstracta que desempenha o papel de "componente" e das suas operações envolvidas, todos os restantes "detalhes" estruturais, comportamentais e consequências estão já especificados no padrão de desenho.

9.7 XMI – XML Metadata Interchange

O XMI (*XML Metadata Interchange*) é o standard da OMG para interoperação de metadata [OMG98]. O XMI foi aplicado inicialmente na metadata de modelação (i.e., de modelos de UML) e de programação, mas está também em curso uma iniciativa para modelar outros domínios de aplicação e de tecnologia tais como *datawarehousing* e componentes.

No respeitante à modelação, o XMI especifica uma estrutura de representação de modelos UML conforme o metamodelo UML. O

principal objectivo do XMI é permitir a interoperação e utilização dos modelos UML de forma independente das plataformas, linguagens, repositórios e ferramentas CASE.

O XMI é definido segundo um extenso documento DTD XML, de 121 páginas. que permite que os modelos sejam representados num formato ASCII, facilmente trocados entre diferentes aplicações e eventualmente legíveis por indivíduos técnicos.

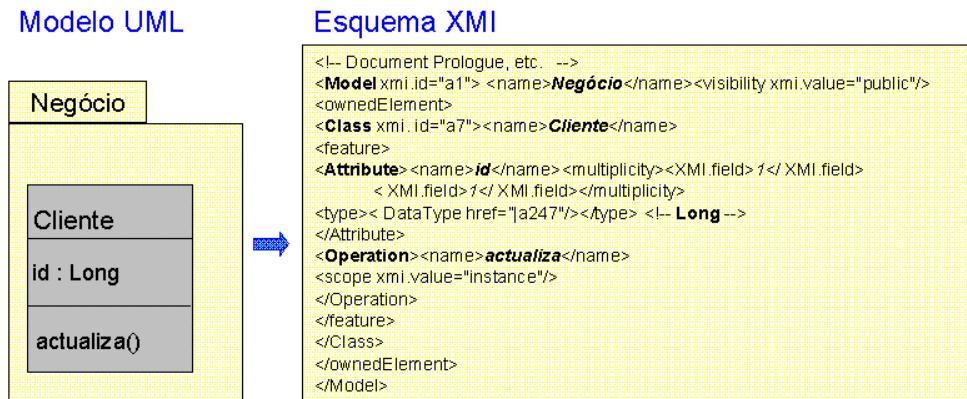



Figura 9.15: Exemplo de um modelo UML representado em XMI.

A Figura 9.15 ilustra, a título de exemplo, um modelo UML constituído pela classe `Cliente` definida no pacote `Negócio`, e a sua respectiva representação em XMI.

9.8 Conclusão

Neste capítulo apresentou-se sucintamente alguns aspectos pouco conhecidos do UML, nomeadamente: (1) a organização e descrição do próprio UML segundo uma arquitectura a quatro camadas; (2) os mecanismos de extensão do UML e o conceito de perfil UML; (3) noções gerais sobre os conceitos de componente, sistemas de componentes e reutilização; (4) tipos parametrizáveis em UML, em particular classes parametrizáveis e colaborações parametrizáveis no contexto de padrões de desenho; e (5) a motivação e contexto prático do XMI enquanto standard para interoperação de modelos UML. Para

além destes aspectos, existem outros que propositadamente não foram tratados neste livro de forma a não estender significativamente o componente da linguagem UML. Enumeram-se, no entanto, alguns desses aspectos avançados que permitem aumentar ainda mais a flexibilidade e potencialidade do UML, designadamente:

- Linguagem OCL (Object Constraint Language).
- Classes de funções utilitárias (*utility classes*).
- Diagramas de colaboração (particularmente, diagramas de sequência) com múltiplos cenários.
- Mapeamento de diagramas de classes em diagramas/esquemas de dados, para definição de bases de dados (ver, a este respeito, as Secções 13.X e 14.Y).
- Mecanismos de definição de estereótipos [Berner99].
- A especificação MOF (*Meta Object Facility*), promovida pela OMG, para definição, manipulação e troca de modelos e meta-modelos em contextos abertos.
- A perspectiva de casos de utilização proposta por Alistair Cockburn [Cockburn00].

Acima de tudo, conclui-se neste capítulo a apresentação e discussão do UML enquanto linguagem agregadora/unificadora (com uma sintaxe e semântica consistente) de várias notações existentes anteriormente nesta área da engenharia.

A apresentação desta parte (Parte 2) seguiu propositadamente uma abordagem didáctica e independente de quaisquer processos ou metodologias de desenvolvimento de software. Ver-se-á nos dois próximos capítulos (Parte 3) como aplicar as diferentes notações UML tendo em conta os próprios processos de desenvolvimento de software.

9.9 Exercícios

Ex.64. Tendo em conta a arquitectura a quatro camadas do UML, diga a que camada pertence cada um dos seguintes elementos: (i) `Class`; (ii) `MetaClass`; (iii) `myServlet002`; (iv) `MyServlet`.

Ex.65. O que é um `Classifier`? Um caso de utilizador é um classificador?

Ex.66. Um estereótipo pode estender um elemento do tipo associação? Dê um exemplo e justifique a sua resposta.

Ex.67. O que são e para que servem os perfis UML?

Ex.68. O que é a classe base de um estereótipo? Qual é a classe base do estereótipo «entity», definido no perfil UML para processos de desenvolvimento de software?

Ex.69. Considere o padrão de desenho COMPOSTO descrito na Secção 9.6.2; aplique-o para modelar a estrutura de elementos compostos e primitivos de um documento XML.

Ex.70. O que é o XMI? Explique a motivação do seu aparecimento.

Parte 3

Metodologias de Desenvolvimento de Software

– *O que é um ritual? – perguntou o Principezinho.*

– *Também é uma coisa que toda a gente se esqueceu – respondeu a raposa. – É o que faz com que um dia seja diferente dos outros dias e uma hora, diferente das outras horas. Os meus caçadores, por exemplo, têm um ritual. À quinta-feira, vão ao baile com as raparigas da aldeia. Assim a quinta-feira é um dia maravilhoso. Eu posso ir passear para as vinhas. Se os caçadores fossem para o baile num dia qualquer, os dias eram todos iguais uns aos outros e eu nunca tinha férias.*

Antoine de Saint-Exupéry, *O Principezinho*.

A Necessidade de uma Metodologia

A Parte 3 marca o retomar do livro na análise dos conceitos mais abrangentes da engenharia de software. Depois de termos apresentado a linguagem UML, descrita ao longo da Parte 2, colocam-se naturalmente, entre outras, as seguintes questões: Como é que usamos o UML para modelar sistemas de software? Quando é que devemos usar os diagramas de interacção, ou de actividades, ou de componentes? Em que fase? Com que nível de detalhe? Que relação existe entre diagramas de casos de utilização e diagramas de interacção? Para auxiliar esta reflexão, considere-se que pedíamos ao leitor para modelar adequadamente, usando o UML, os casos de estudo apresentados nas Secções 10.8 e 11.4. O leitor, por certo se sentiria desconfortável com este pedido. Porquê? Porque não sabe ainda que passos deve seguir, que tipos de diagramas deve usar em cada passo, como relacioná-los, etc. Ou seja porque precisa conhecer e aplicar uma determinada metodologia que o ajude no processo de desenvolvimento (ou simplesmente de modelação) de software.

O que é uma Metodologia?

Tal como discutido no Capítulo 2, o processo de desenvolvimento de software consiste genericamente num conjunto de fases, tarefas e actividades, realizadas por intervenientes que desempenham várias funções de modo a elaborarem diversos artefactos e que conjuntamente contribuem para a produção de um sistema de software. A metodologia adiciona a este conceito a noção de técnicas, notações e utilização de ferramentas.

O processo de desenvolvimento de software é um acto complexo, com múltiplas variáveis e inúmeras vezes com qualidade reduzida, ultrapassando os prazos e os orçamentos previstos. Não há receitas milagrosas, por isso um processo deve ser adaptado e configurado conforme o perfil das empresas, dos projectos e dos intervenientes envolvidos. Não acreditamos na produção de software em massa e automatizada; é e será sempre uma combinação entre engenharia e arte.

Nesta terceira parte do livro, aprofundamos a problemática das metodologias de desenvolvimento de software através da análise e discussão de duas propostas concretas que adoptam o UML como linguagem de modelação: o RUP e o ICONIX. De referir que os autores destas propostas classificam-nas como processos e não como metodologias, mas remetemos o leitor para o Capítulo 2 para esclarecimentos sobre este assunto.

Organização da Parte 3

O Capítulo 10, "Metodologia RUP", apresenta o *Rational Unified Process*, uma metodologia completa aplicada ao desenvolvimento de software. Encontra-se fortemente integrado com o UML, ou não tivesse resultado da continuação dos esforços dos mesmos autores. O RUP segue um modelo iterativo e incremental, centrado numa arquitectura e conduzido por casos de utilização. É uma metodologia que se baseia no paradigma da orientação por objectos, e a abrangência das suas propostas contempla a possibilidade de adaptação à realidade de cada projecto ou organização.

Neste capítulo, apresenta-se uma panorâmica geral do RUP, em particular as suas diferentes visões: a visão dinâmica (baseada em ciclos, fases e iterações) e a visão estática (baseada em *workflows*, actividades e artefactos). Por fim, discute-se o RUP através da sua aplicação a um caso prático, de dimensão e complexidade não trivial.

O Capítulo 11, "Metodologia ICONIX", apresenta a metodologia ICONIX, para desenvolvimento de software segundo uma abordagem muito prática, algures entre a complexidade e abrangência do RUP e a simplicidade e o pragmatismo do Extreme Programming. O ICONIX é conduzido por casos de utilização, iterativo e incremental, tal como o RUP, mas sem a complexidade que este preconiza. Por outro lado, é relativamente pequeno e simples, tal como o XP, mas sem eliminar as tarefas de análise e de desenho que aquele não contempla. Discute-se o ICONIX através da sua aplicação na modelação do caso de estudo WebDEI, o qual consiste num sistema de informação Web a quatro camadas, desenvolvido exclusivamente com tecnologia Java.

Capítulo 10 - METODOLOGIA RUP

Tópicos

- Introdução
- Enquadramento
- Características Principais
- As 4+1 Visões
- Visão Geral
- Ciclos, Fases e Iterações - A Componente Dinâmica
- *Workflows*, Actividades e Artefactos – A Componente Estática
- Enunciado do Caso de Estudo DGD
- Resolução do Caso de Estudo DGD
- Conclusão
- Exercícios

10.1 Introdução

Durante a década de 90, tornou-se óbvia para muitos teóricos da engenharia de software a vantagem do paradigma da orientação por objectos e, nesse sentido, começaram a proliferar as metodologias que tinham por base os respectivos conceitos. O exagerado número de metodologias que entretanto foram propostas fez recordar a experiência negativa que já tinha ocorrido com as metodologias estruturadas, pelo que ganhou força a ideia da criação de uma metodologia de desenvolvimento de software "unificada", que tirasse partido da experiência e dos conceitos da linguagem UML. Assim, a ideia de uma meto-

dologia e processo unificados foram mais uma consequência natural da aproximação dos “três amigos” e que já tinha conduzido à definição do UML.

Baseado na noção de metodologia unificada existem actualmente várias propostas de metodologias de desenvolvimento de software, umas mais teóricas, abrangentes e completas, como é o caso do USDP (*Unified Software Development Process*) [Jacobson99] e o RUP (*Rational Unified Process*) [Kruchten00]; outras mais práticas e simples como é o caso do ICONIX [Rosenberg99] ou do GRASP [Larman98].

A forte integração com o UML é notória ao longo de todo o processo, mas importa referir que o UML e o RUP têm âmbitos de intervenção distintos: o UML é uma linguagem de notação e de modelação, enquanto que o RUP é uma metodologia de desenvolvimento completa. A analogia fornecida por Philippe Kruchten [Kruchten00] permite esclarecer melhor a importância do RUP: o UML funciona como um conjunto de palavras e de regras que definem a forma como devem ser combinadas para formar frases. No entanto, se quisermos escrever um livro, é necessário outro tipo de regras, nomeadamente relativas à formatação de capítulos, à criação de um índice, à identificação de palavras chave e à pesquisa de bibliografia. É esta a função que o RUP representa.

O RUP é uma metodologia (se bem que os seus autores a designem como “processo” – veja-se a Secção 2.2 para mais esclarecimentos) de engenharia de software desenvolvida e comercializada pela empresa Rational Software. Tendo em conta que a construção de software de qualidade de uma forma repetitiva e previsível é difícil, e que as causas dos problemas associados a este tipo de desenvolvimento têm sido uma constante ao longo do tempo, o RUP propõe várias boas práticas (referidas na Secção 2.4) para o desenvolvimento de software e aplica-as de forma integrada.

O RUP é mais do que uma "simples" metodologia de desenvolvimento de software, uma vez que pode funcionar como um conjunto de princípios genéricos utilizado para instanciar e configurar várias metodologias concretas, conforme o tipo de organização, o domínio de aplicação, o nível de competências, etc.

O RUP, enquanto produto, é apresentado resumidamente no livro *The Rational Unified Process – An Introduction* [Kruchten00] e de forma detalhada numa *knowlegde base* disponível *online* (www.rational.com/products/rup/index.jsp) ou em CD-ROM, ambos comercializados pela Rational. O produto RUP encontra-se integrado com ferramentas da mesma empresa que se destinam a suportar outras actividades do processo de desenvolvimento, tais como gestão de requisitos (RequisitePro), controle das alterações (ClearQuest), modelação do sistema (Rose). Como qualquer outro produto de software, são produzidas frequentemente novas versões de forma a acompanhar a evolução do mercado.

Na Figura 10.1 podemos observar o aspecto da interface do produto RUP, que apresenta semelhanças com o Microsoft Explorer, o que é compreensível: uma vez que se trata de uma ferramenta que disponibiliza uma quantidade de informação elevada, as suas funcionalidades deverão estar optimizadas para facilitar a pesquisa dessa mesma informação.

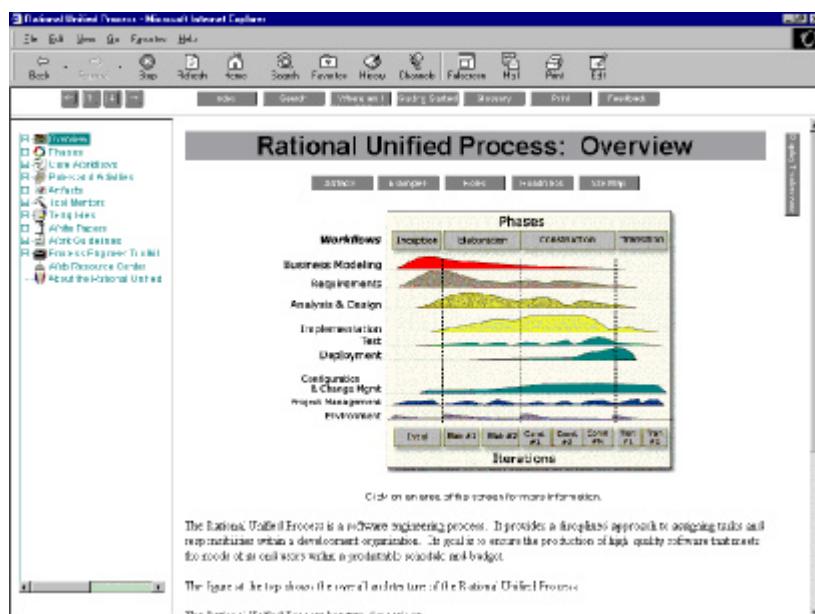



Figura 10.1: Aspecto gráfico do RUP.

Para além da sua versão base, encontram-se disponíveis variantes já adaptadas para situações concretas do desenvolvimento de software; por exemplo, para pequenos projectos ou projectos na área do negócio electrónico. São também incluídos *templates* para ligação a outras ferramentas da Rational ou de outros fabricantes, de modo a facilitar a produção dos artefactos do RUP.

Este capítulo descreve o RUP enquanto metodologia de desenvolvimento. Contudo, dada a sua complexidade e dimensão, são apresentadas as ideias principais de forma resumida e são demonstrados alguns aspectos concretos da sua aplicação, através de um caso prático, concentrando a análise nas actividades realizadas. Aborda-se superficialmente a documentação e modelos produzidos, já referidos extensamente noutros capítulos, que por isso não são aqui apresentados.

10.2 Enquadramento

O RUP, apesar de existir há muito pouco tempo (mais concretamente desde 1998), tem na sua origem ideias e experiências que remontam há mais de trinta anos, nomeadamente nas abordagens seguidas na Ericsson, onde trabalhou Ivar Jacobson. Já em 1967 esta empresa modelava todo o seu sistema em vários blocos relacionados, construídos numa estrutura hierárquica, utilizando conceitos muito próximos do UML [Jacobson85]. No entanto, as abordagens com maior influência directa no RUP surgiram a partir de finais da década de 80, como ilustrado na Figura 10.2.

Figura 10.2 : Evolução do RUP.

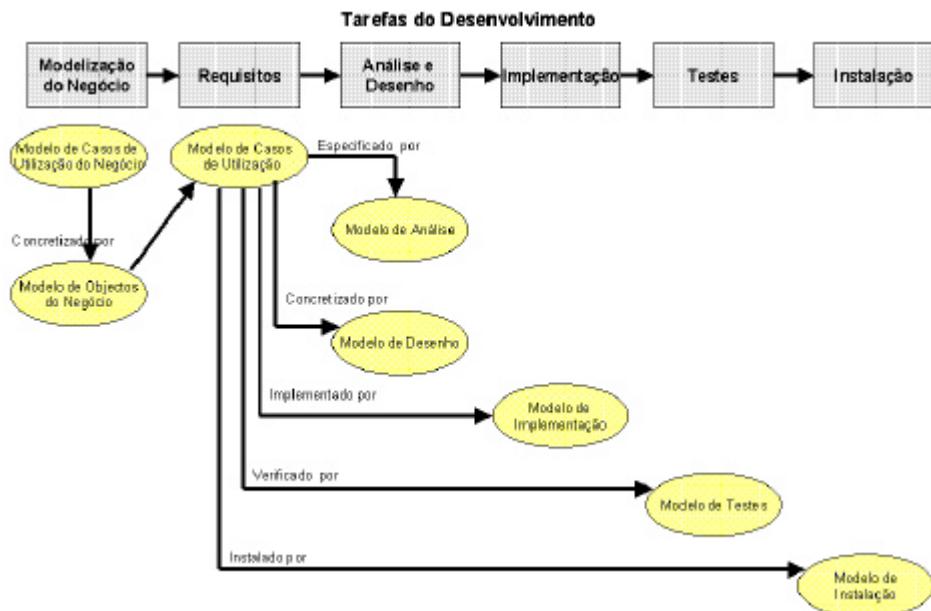
Em 1987, Ivar Jacobson deixou a Ericsson e fundou a empresa Objectory AB, onde até 1995 desenvolveu o *Objectory Process*, altura em que esta empresa foi adquirida pela Rational. Para além de outros conceitos de análise e desenho orientado por objectos, foi no contexto do *Objectory Process* que pela primeira vez se definiu o conceito casos de utilização [Jacobson92].

Pelo seu lado, a Rational vinha desenvolvendo desde 1981, e com maior ênfase a partir do início da década de 90, um conjunto de iniciativas com o objectivo de conceber um ambiente interactivo que permitisse aumentar a produtividade do desenvolvimento de sistemas complexos [Archer86]. As ideias mais significativas tinham a ver com a necessidade da aplicação de um processo iterativo e a definição de uma arquitectura baseada em componentes. Estas iniciativas conduziram à elaboração da *Rational Approach*, sendo Philippe Kruchten um dos seus principais mentores.

A proliferação das metodologias baseadas nos conceitos da orientação por objectos e de desenvolvimento iterativo tornou óbvia a necessidade de uniformizar as mesmas, o que foi facilitado com a entrada para a Rational de alguns dos principais "teóricos" das metodologias existentes na altura, nomeadamente a dos "três amigos" (Grady Booch, James Rumbaugh e Ivar Jacobson) e a conjugação das ideias das suas abordagens (*Booch Method*, *Object Modeling Technique* e *Objectory* respectivamente) com as da Rational.

Esta integração deu origem ao que na altura se designou por Rational *Objectory Process*, cuja designação, a partir de 1998, passou a ser *Rational Unified Process,* sobretudo devido ao número de contribuições adicionais que entretanto foram integradas (e que mesmo os principais autores indicam não saber ao certo quantas e quais foram!). Estas contribuições abrangem áreas tão diversas como a gestão de requisitos (incorporada com a aquisição da empresa Requisite) e o controle de qualidade (com a aquisição da SQA).

Com a integração das ideias propostas no UML 1.3 e a inclusão de especificidades de desenvolvimento para as tecnologias Internet, o RUP evoluiu para a versão 5.5 em 1999, tendo sido disponibilizada durante o

ano 2000 a versão RUP 2000. Esta é a versão do produto (e da metodologia) que iremos analisar neste capítulo.

O RUP é hoje em dia utilizado por diversas organizações, de várias indústrias e em projectos de dimensão variável, com particular destaque em projectos de negócio electrónico. Segundo [Kruchten00], em finais de 1999 mais de mil organizações de grande dimensão tinham já adoptado o RUP.

10.3 Características Principais

O RUP suporta diversas boas práticas do desenvolvimento de software em particular aquelas que foram identificadas no Capítulo 2: (1) é uma metodologia de desenvolvimento de software iterativa; (2) propõe a gestão integrada de requisitos desde a sua identificação até à implementação; (3) propõe o desenvolvimento de software baseado em arquitecturas de software e em componentes; (4) defende a modelação visual; e (5) o controle de qualidade permanente. Para além destas características, o RUP integra outras ideias fundamentais, nomeadamente o facto de ser orientado por casos de utilização.

10.3.1 Metodologia Conduzida por Casos de Utilização

No Capítulo 5 analisámos como um sistema pode ser especificado, na perspectiva dos seus utilizadores, através de casos de utilização, pelo facto destes diagramas permitirem capturar os requisitos funcionais.

Figura 10:3 : Metodologia Conduzida por Casos de Utilização.

No caso do RUP, tal como noutras metodologias que recorrem ao UML, a inovação reside no facto dos casos de utilização, para além de especificarem os requisitos do sistema, conduzirem também o seu próprio desenho, implementação e teste, ou seja, todo o processo de desenvolvimento. Dito de outra forma, significa que o processo de desenvolvimento segue um fluxo, em que os casos de utilização são especificados, desenhados, implementados, e no fim são a fonte a partir dos quais os testes são definidos e realizados. A Figura 10.3 ilustra este facto: todos os modelos construídos ao longo das diversas tarefas do RUP são elaborados a partir do modelo de casos de utilização, produzido durante a tarefa de levantamento de requisitos.

Embora seja verdade que os casos de utilização conduzem a metodologia, eles não são seleccionados ao acaso e de forma isolada: os casos de utilização baseiam-se na arquitectura do sistema, e a arquitectura influencia a definição e selecção dos casos de utilização, segundo uma aproximação iterativa e incremental.

10.3.2 Metodologia Centrada numa Arquitectura

A função da arquitectura de software é reflectir decisões sobre a organização do sistema de software, nomeadamente:

- Os elementos estruturais (e suas interfaces) que compõem o sistema.
- O seu comportamento especificado através de colaborações entre eles.
- A composição dos elementos estruturais e de comportamento em subsistemas progressivamente maiores.
- O estilo arquitectural, que guia esta organização, os elementos e suas interfaces, as suas colaborações e a sua composição.

A arquitectura de software decide ainda sobre restrições e compromissos relativos a diferentes aspectos do sistema, tais como: utilização, funcionalidades, desempenho, tolerância a alterações, reutilização, compreensão, economia e estética.

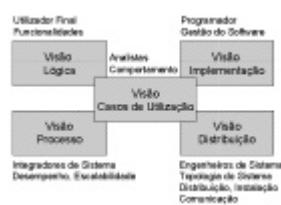
A arquitectura de software é representada fisicamente através de um conjunto de visões sobre distintos tipos de modelos, elaborados quer em função dos intervenientes e destinatários, quer do momento em que são produzidos. Assim, podemos ter a visão do modelo de casos de utilização; a visão do modelo de análise; a visão do modelo de desenho; a visão do modelo de implementação; e a visão do modelo de instalação. Estas diferentes visões evidenciam e apresentam o "estilo arquitectural" que o sistema deverá apresentar.

A arquitectura de software permite implementar os casos de utilização requeridos, com uma adequada relação custo-benefício, tendo em conta a experiência acumulada (por exemplo, anteriores arquitecturas e padrões arquitecturais) e um conjunto definido de restrições conforme ilustrado na Figura 10.4.

Figura 10.4: Os condicionalismos de uma arquitectura de software.

Por outro lado, os casos de utilização são desenvolvidos com base na informação fornecida pelos clientes e utilizadores, mas também são fortemente condicionados pela arquitectura subjacente que exista ou que venha a existir (Figura 10.4).

Figura 10.5: As condicionantes dos casos de utilização.

10.3.3 Metodologia Iterativa e Incremental

Na Secção 2.6 discutimos as diferenças existentes entre os processos que implementavam um modelo em cascata e os processos iterativos e incrementais. As metodologias unificadas sugerem uma aproximação iterativa e incremental para projectos de média a grande dimensão, reduzindo os de pequena dimensão a apenas uma iteração. A justificação para esta proposta é o facto de um projecto ser mais facilmente gerido e executado se for dividido em várias partes ou mini-projectos, em que cada mini-projecto é uma iteração que resulta num incremento, que deverá ser devidamente planeado, controlado e executado (através de análise, desenho, implementação e testes).

Um aspecto importante é a identificação do âmbito e do detalhe do trabalho a ser realizado em cada iteração, o que depende de dois factores: por um lado, os casos de utilização seleccionados para cada iteração; por outro lado, os factores de risco mais importantes que deverão ser analisados e resolvidos o mais cedo possível.

Por isso, em cada iteração os participantes devem identificar e especificar os casos de utilização relevantes; efectuar o desenho com base numa determinada arquitectura de software; implementar o desenho através de um conjunto de componentes; e verificar se os componentes estão em conformidade com as especificações dos casos de utilização. Note-se que estas três características do RUP – conduzido por casos de utilização, centrado numa arquitectura e iterativo e incremental – estão eminentemente relacionadas.

10.4 As 4+1 Visões do RUP

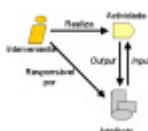
Existem diferentes perspectivas segundo as quais o processo de desenvolvimento de software e o produto dele resultante podem ser encarados, sobretudo tendo em conta os interesses particulares dos diversos intervenientes no processo, que resultam das suas funções na organização. A definição de uma arquitectura sólida é a chave de sucesso para qualquer projecto que utilize uma abordagem orientada por objectos, segundo Booch [Booch95]. O RUP apresenta um modelo segundo o qual considera a existência de 4+1 visões do sistema, definidas ao longo do processo, e cuja representação pode ser observada na Figura 10.6. Cada visão é concretizada através de diversos diagramas do UML.

Figura 10.6: As 4+1 Visões do RUP.

Para os utilizadores o mais importante é a satisfação dos seus requisitos funcionais, expressos numa visão lógica (*Logical View*) do sistema. A este nível é necessário garantir que se tem informação sobre o que o sistema deve fazer. A arquitectura lógica do sistema é representada através dos diagramas de classes que modelam as principais abstracções do sistema (classes, relações e pacotes).

Para os intervenientes técnicos que participam no desenvolvimento do sistema, interessa ter a visão da organização do mesmo em termos dos módulos de software (desde o código fonte até aos executáveis), bem como resolver questões relacionadas com a gestão da configuração, a facilidade do desenvolvimento, a reutilização e restrições relacionadas com as linguagens de programação. Esta é a visão de implementação (*Implementation View*), que reflecte a estrutura estática do sistema. Os principais elementos de modelação nesta visão incluem os pacotes e os diagramas de componentes.

A visão de processamento (*Process View*) do sistema preocupa-se em representar os conceitos relacionados com a implementação do sistema no seu ambiente de produção. Estamos a falar de conceitos como tarefas, actividades e processos, bem como com as suas interacções. Nesta visão, são consideradas questões como paralelismo e concorrência na execução, tolerância a falhas, distribuição de objectos, tempos de resposta e escalabilidade, desempenho, fiabilidade e integridade do sistema. Esta visão é modelada por diagramas de componentes.

A visão de instalação (*Deployment View*) representa a correspondência entre os componentes desenvolvidos pelo projecto e o respectivo suporte tecnológico, modelando a configuração dos elementos de processamento em tempo de execução. As principais preocupações são questões como a migração e instalação do sistema e o respectivo desempenho, disponibilidade, fiabilidade e escalabilidade. São utilizados diagramas de componentes e de instalação, que permitem visualizar a localização física de componentes na infra-estrutura física e/ou computacional da organização.

Finalmente, a visão dos casos de utilização (*Use Case View*) tem em consideração os casos de utilização importantes, de forma a definir, por um lado, a arquitectura nas fases iniciais do processo e, por outro, a validar mais tarde a percepção das restantes visões. Funciona assim como a visão integradora de todas as restantes, e reforça a ideia do RUP enquanto metodologia conduzida por casos de utilização, que são naturalmente os principais diagramas de modelação desta visão.

10.5 Visão Geral

10.5.1 Conceitos Gerais

Uma metodologia, de acordo com a definição apresentada no Capítulo 2, deve definir **quem** faz o **quê**, **quando** e **como**. Estas quatro palavras-chave correspondem a alguns conceitos base utilizados pelo RUP, que importa desde já clarificar:

- Um **interveniente** (quem), que na terminologia RUP se designa por *worker:* conceito que define o comportamento e as responsabilidades de um ou mais indivíduos. Como exemplos, temos o analista de sistemas, o programador, o elemento que efectua os testes.

- Uma **actividade** (como): unidade de trabalho que um interveniente pode ser chamado a realizar. A sua granularidade varia de algumas horas até vários dias e é a unidade de planeamento mais elementar, se bem que, do ponto de vista de acções a executar, podem ser ainda divididas em passos mais elementares. Como exemplos de actividades podemos indicar o planeamento de uma iteração ou a identificação de casos de utilização e de actores.

- Um **artefacto** (o quê): qualquer informação que é produzida, alterada ou utilizada por um ou mais intervenientes durante uma actividade. Como exemplo de artefactos temos os modelos, documentos, planos, código fonte e executáveis. Cada artefacto pode ser elaborado com o apoio de ferramentas distintas.

- Uma **tarefa** (quando), que na terminologia RUP se designa por *workflow:* é uma sequência de actividades que produz um resultado observável. Utilizando a notação UML, um *workflow* é normalmente representado por um diagrama de actividades.

Figura 10.7: Conceitos importantes do RUP.

A arquitectura do RUP encontra-se estruturada segundo duas dimensões, que reflectem as duas visões através das quais um sistema pode ser descrito (ver Figura 10.8)

- Componente dinâmica que reflecte a evolução temporal de um projecto, expressa em ciclos, fases, iterações e objectivos a atingir.
- Componente estática que reflecte quais as tarefas (*workflows*) e actividades realizadas, os *outputs* produzidos (artefactos), e os intervenientes.

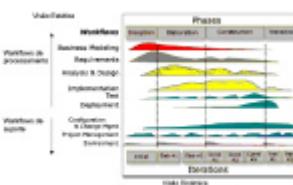
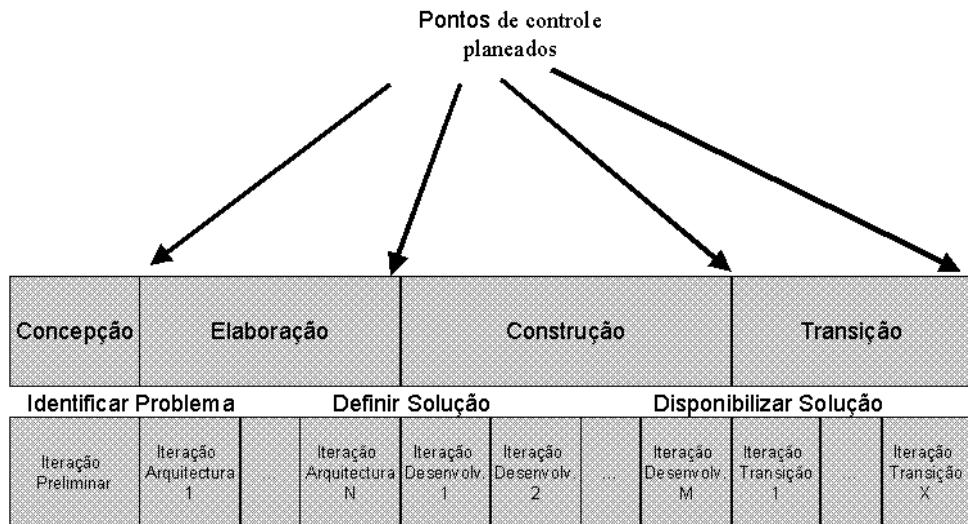



Figura 10.8: As dimensões do RUP.

10.5.2 Componente Dinâmica

Quando se considera a componente dinâmica do RUP, podemos encarar um projecto como uma sequência de fases, cada qual com várias iterações. Segundo o RUP, as fases são as seguintes:

- Concepção (*Inception*): pretende obter uma visão global do sistema, eliminar os riscos mais importantes e efectuar a definição do âmbito do projecto.
- Elaboração (*Elaboration*): tem por objectivo especificar as funcionalidades e desenhar a arquitectura.
- Construção (*Construction*): pretende implementar e testar o software.
- Transição (*Transition*): pretende distribuir o produto ao cliente final, bem como efectuar todas as actividades necessárias (e.g., formação) para garantir o respectivo sucesso.

Cada fase é concluída com um ponto de controle (*milestone*) bem definido, no qual os resultados obtidos são comparados com objectivos principais, e onde devem ser tomadas algumas decisões críticas. As mesmas actividades são executadas repetidamente em diversas

iterações, cada uma aumentando o nível de detalhe e/ou de abrangência.

Figura 10.9: Fases, iterações e milestones.

10.5.3 Componente Estática

A visão estática do RUP descreve as tarefas (*workflows*) e as actividades realizadas, bem como os artefactos produzidos e os respectivos intervenientes. Esta visão encontra-se centrada no conceito de *workflow* acima descrito. No RUP os *workflows* encontram-se divididos em *workflows* de processamento e *workflows* de suporte. No primeiro grupo incluem-se os *workflows* de:

- Modelação de Processos de Negócio (*Business Modeling*): inclui as actividades necessárias para compreender os processos de negócio da organização.
- Requisitos (*Requirements*): agrupa o conjunto de actividades relacionadas com a identificação e modelação dos requisitos do sistema.
- Análise e Desenho (*Analysis and Design*): actividades de definição do modo como o sistema será construído na fase de implementação.

- Implementação (*Implementation*): actividades de produção de código e testes unitários.
- Testes (*Test*): actividades de verificação de todo o sistema.
- Instalação (*Deployment*): actividades de disponibilização do sistema para os seus utilizadores finais.

Os *workflows* de suporte agrupam actividades que decorrem ao longo de todo o projecto, que não contribuem directamente para o produto final, mas são essenciais para garantir o respectivo sucesso, como sejam a gestão do projecto (*Project Management*), a gestão da configuração e das alterações (*Configuration and Change Management*) e a definição do ambiente de desenvolvimento (*Environment*).

É de realçar a forma como os conceitos fase, tarefa e actividade se encontram relacionados no RUP comparativamente às metodologias tradicionais. Nestas últimas, a relação entre estes três conceitos é claramente de natureza hierárquica e sequencial: uma fase é um sequência de tarefas, as quais são uma sequência de actividades. Por exemplo, a fase de concepção inclui a tarefa de planeamento, que tem, entre outras, a actividade de elaboração do plano do projecto. Em nenhuma circunstância, nas metodologias tradicionais, podemos ter situações em que uma tarefa se prolonga durante várias fases.

A visão do RUP, conforme descrito, é inovadora, indo contra as formas mais instituídas e tradicionais de pensar o processo de desenvolvimento de software. Uma fase decorre em várias iterações, o que desde logo introduz algumas diferenças relativamente às abordagens tradicionais. No entanto, a principal inovação resulta do facto de uma tarefa se estender, ao longo do tempo, por várias fases, e por conseguinte as mesmas actividades de uma tarefa são repetidas em várias iterações.

10.6 Ciclos, Fases e Iterações - A Componente Dinâmica

Diz-se que esta é a visão dinâmica do RUP porque ela agrupa os com-ceitos que representam a evolução de um projecto ao longo do tempo.

Em termos da hierarquia de conceitos, o mais abrangente de todos é o **ciclo**, que pode ser definido como uma sequência das quatro fases do RUP (concepção, elaboração, construção e transição). Um ciclo produz sempre uma versão funcional disponibilizável para o cliente final. Se, nalguns casos, os projectos poderão apresentar apenas um ciclo, é normal que logo no início de um projecto seja possível ou mesmo necessário prever a ocorrência de vários ciclos:

- No desenvolvimento de sistemas complexos, vale a pena ir disponibilizando para os utilizadores finais versões incrementais de funcionalidades.
- No caso de projectos de software comercial, os ciclos subsequentes representarão a tarefa de manutenção correctiva e evolutiva, que, no fundo, não é mais do que a aplicação de todas as actividades já realizadas no primeiro ciclo, algumas com intensidade diferente.

Uma **fase** é o período de tempo que decorre entre dois pontos de controle planeados (*milestones*), durante o qual um conjunto bem definido de objectivos é alcançado, são produzidos artefactos, e são tomadas decisões sobre a continuidade do projecto.

Do ponto de vista formal, não são especificadas actividades relativamente às fases, mas sim aos *workflows*, que são parte integrante da outra visão, a visão estática. Relativamente a cada fase, são especificados o conjunto de objectivos a atingir através da execução de actividades de um ou mais *workflows*, ao longo de uma ou mais iterações, e os respectivos critérios de avaliação. Nesta perspectiva, é óbvio que durante o período de tempo de uma fase ocorrerão actividades, mas esta relação é indirecta, através dos respectivos *workflows*.

Esta é uma visão muito mais natural e realista, visto que é normal que a mesma actividade seja ciclicamente realizada ao longo de um processo, nomeadamente:

- A identificação e modelação de requisitos ocorre sobretudo no início do processo, mas pode também ser efectuada já durante a implementação (como todas os elementos que estão ligados à área de desenvolvimento de sistemas de informação sabem que é assim que acontece na realidade).
- A codificação pode ser realizada desde o início das tarefas de análise, para a elaboração de protótipos e análise de riscos prévia.

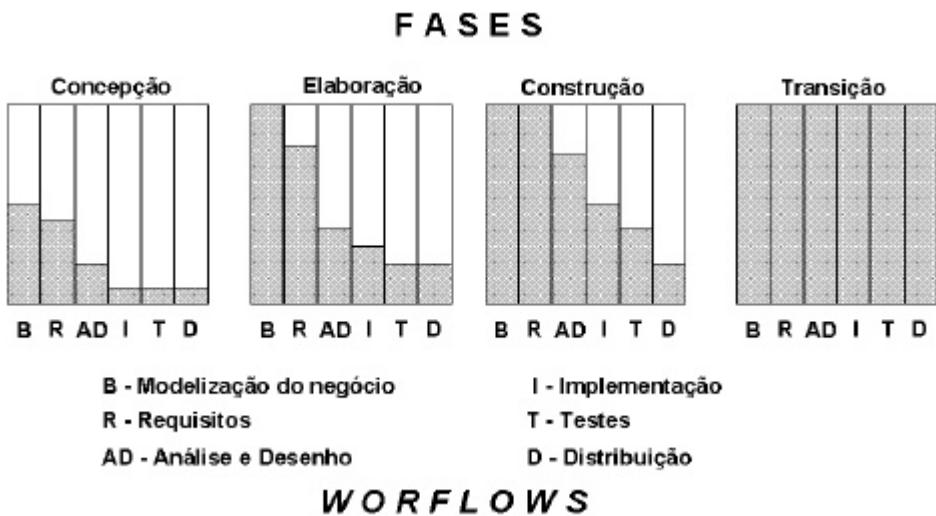



Figura 10.10: Percentagem de finalização de cada workflow no final de cada fase.

Cada fase pode ser realizada em diversas **iterações**, cada uma constituindo-se como uma sequência de actividades, que resulta numa versão interna ou externa (no caso da última iteração da fase de transição) de um produto executável, e que cresce incrementalmente ao longo das várias iterações. Com esta abordagem, os riscos são resolvidos antecipadamente à tomada de decisões da arquitectura, as alterações são geridas mais facilmente e a reutilização de artefactos é mais elevada, o que possibilita uma qualidade global superior.

Analisamos de seguida cada uma das fases do RUP.

10.6.1 Concepção

Durante a fase de concepção, os objectivos são: (1) definir o *business case* (justificação de negócio) do projecto, incluindo questões como os critérios de sucesso, a análise de risco, os recursos necessários e (2) elaborar um plano das fases seguintes com os principais pontos de decisão. É ainda nesta fase que é delimitado o âmbito do projecto e são identificadas todas as entidades externas com as quais o sistema inte-

rage. Todos os casos de utilização devem ser identificados e os aparentemente mais relevantes deverão ser descritos.

Os principais resultados a obter no final desta fase são:

- Documento com a visão geral do problema: requisitos principais, funcionalidades mais significativas, restrições importantes.
- Modelo de casos de utilização, dos quais tipicamente entre 10% a 20% do total deverão estar especificados.
- Glossário inicial do projecto.
- Relatório inicial de avaliação do risco.
- Justificação de negócio (*business case*) inicial.
- Plano de projecto.
- Protótipos iniciais.

Cada fase tem critérios de avaliação, a aplicar no seu final, de modo a confirmar a realização dos objectivos. No caso da fase de concepção, estes incluem:

- Aprovação dos patrocinadores e interessados no sistema relativamente ao âmbito e às estimativas de custos e prazos elaboradas.
- Compreensão dos requisitos evidenciada pela equipa do projecto, através dos primeiros casos de utilização elaborados.
- Credibilidade das estimativas de custos e prazos e do processo de desenvolvimento definido.
- Adequabilidade (ao nível da abrangência e do detalhe) dos protótipos da arquitectura desenvolvidos.
- Comparação entre gastos planeados e efectivamente realizados.

10.6.2 Elaboração

A fase de elaboração tem como preocupação central analisar o domínio do problema detalhadamente, de forma a definir uma arquitectura de componentes do sistema sólida e robusta e a pormenorizar o plano do projecto, em resultado do conhecimento entretanto adquirido. Com estes objectivos atingidos, é possível identificar as medidas que permitam eliminar os riscos mais significativos. Em termos práticos, é ainda objectivo desta fase construir o protótipo da arquitectura (numa ou

mais iterações da fase) e identificar todos os actores e casos de utilização, descrevendo a sua grande maioria. Por isso, alguns dos principais resultados a obter nesta fase são:

- Modelo de casos de utilização (considera-se que pelo menos 80% do total dos casos de utilização deverão estar especificados).
- Requisitos totais do sistema, incluindo os funcionais e não funcionais.
- Descrição da arquitectura de software.
- Protótipo executável da arquitectura.
- Lista de riscos e *business case* revistos.
- Plano de projecto revisto.
- Manual do utilizador (preliminar).

No final desta fase são examinados o âmbito e os objectivos detalhados do sistema, é seleccionada a arquitectura e são identificadas medidas para solucionar (ou pelo menos atenuar) os principais riscos. Os critérios de avaliação para verificar a satisfação dos resultados face aos objectivos incluem:

- Estabilidade da visão do produto e da arquitectura.
- Resolução dos principais riscos do sistema.
- Detalhe e estabilidade do planeamento para a fase de construção e credibilidade das estimativas efectuadas.
- Aprovação pelos interessados no sistema que a percepção actualmente existente do mesmo por parte da equipa do projectos, pode ser atingida com o plano do projecto e no contexto da arquitectura proposta.
- Comparação entre gastos planeados e efectivamente realizados.

10.6.3 Construção

É objectivo principal da fase de construção efectuar o desenvolvimento e integração de componentes no produto final, bem como testes exaustivos a todas as funcionalidades. É prestada particular atenção ao controle de custos, prazos e qualidade do produto. No final desta fase existe um produto que pode ser disponibilizado aos utilizadores, que

inclui os diversos componentes de software integrados e instalados nas plataformas identificadas.

De modo a garantir que todos os componentes tecnológicos estão prontos para a entrada em produção, sem riscos elevados, e que o produto se encontra suficientemente estável e maduro para ser utiliza-do. Mais uma vez, é também necessário comparar os gastos planeados e os efectivamente realizados, de forma a identificar e analisar eventuais desvios.

10.6.4 Transição

É objectivo desta fase efectuar a disponibilização do software para o utilizador. Para que tal aconteça, é necessário garantir a conformidade do sistema produzido relativamente aos requisitos inicialmente definidos, que o conjunto de funcionalidades identificadas tenha sido desenvolvido e que o seu todo constitui um sistema funcional; é ainda necessário garantir que a documentação do utilizador foi elaborada. Por isso, o sistema deve ser validado em relação às expectativas dos utilizadores (o que pode envolver a realização de processamentos paralelos), as bases de dados operacionais devem ser convertidas, os utilizadores devem receber formação e finalmente o produto deve ser instalado ou distribuído. Os principais critérios de avaliação são a satisfação do utilizador e a comparação entre as estimativas inicial-mente previstas e o que efectivamente foi realizado.

10.6.5 Comentários Gerais

A abordagem iterativa é fácil de representar mas não de aplicar. O processo iterativo encontra-se organizado em fases, mas ao contrário das abordagens tradicionais é ortogonal às tarefas que compreende. A aplicação destas quatro fases constitui o que se designa por um **ciclo de desenvolvimento**, o qual produz uma versão do software disponibilizável para o cliente. Cada produto terá sempre um **ciclo inicial**, e a não ser que o desenvolvimento pare (o que é pouco provável, quanto mais não seja porque se entra na fase de

manutenção), poderão existir posteriores **ciclos de evolução**, que produzem novas versões.

Para além dos ciclos e das fases, cada uma das fases pode ainda ser dividida em iterações, nas quais se aplicam sucessivamente actividades que estão previstas nos vários *workflows* (ver Secção 10.7), com graus de intensidade diversos consoante a fase e a iteração. Assim, na iteração da fase de concepção (tipicamente apenas uma), procura-se obter uma visão global dos requisitos e determinar o âmbito e o esforço de trabalho a realizar. Nas iterações da fase de elaboração, o principal objectivo é identificar os requisitos, e por isso as principais actividades pertencem ao *workflow* de requisitos. No entanto, este objectivo da fase de elaboração é também atingido com actividades do *workflow* de análise e desenho e do *workflow* de implementação, como sejam a produção de protótipos e a eliminação de riscos técnicos através da selecção de alternativas.

Nas diversas iterações da fase de construção, e uma vez que o objectivo é construir um primeiro produto operacional, as actividades do *workflow* de implementação vão aumentando de importância relativa, o que não significa que algumas decisões sobre a arquitectura dos componentes não tenham que ser revistas em função de problemas só identificados nesta fase, e daí existirem também actividades dos *wokflows* de requisitos e de desenho. Finalmente, as iterações da fase de transição procuram disponibilizar o produto para o seu utilizador final, sendo por isso natural que as actividades principais sejam dos *workflows* de testes e de instalação. Contudo, as correcções a efectuar podem implicar ainda a revisão da estrutura do sistema (actividades do *workflow* de desenho) e a codificação de componentes (actividades do *workflow* de implementação).

Esta abordagem iterativa permite que os riscos sejam identificados e controlados atempadamente no processo de desenvolvimento, que as alterações sejam de facto melhor controladas, que a reutilização dos diversos artefactos, a qualidade do produto e a produtividade dos técnicos informáticos sejam superiores.

10.7 *Workflows*, Actividades e Artefactos – A Componente Estática

A especificação da estrutura estática do RUP envolve a descrição de quem (Interveniente - *Worker*) faz o quê (Artefactos), como (Actividades) e quando (*Workflows*). O modelo do RUP prevê a existência de seis *workflows* principais, que agrupam as actividades directamente relacionadas com a produção do produto final:

- Modelação do negócio
- Requisitos
- Análise e Desenho
- Implementação
- Testes
- Instalação.

Prevê ainda a existência de três *workflows* de suporte, que incluem actividades de apoio executadas ao longo de todo o processo, que são essenciais para garantir o sucesso do mesmo:

- Gestão de projectos
- Configuração e gestão de alterações
- Definição do Ambiente

Ao contrário das abordagens tradicionais, estes *workflows* não só não são sequenciais, como são repetidos diversas vezes ao longo das várias fases do processo, em várias iterações, possivelmente realizando actividades diferentes ou com um grau variável de intensidade e detalhe.

Podemos dizer que o *workflow* completo de um projecto inclui estes nove *workflows* individuais, repetidos com ênfase e intensidade variável ao longo de cada fase e iteração. A realização das várias actividades produz diversos artefactos, os mais significativos dos quais são apresentados na Figura 10.11.

O RUP tem um conjunto muito extenso de diagramas para especificar as actividades realizadas por cada interveniente e os artefactos produzidos, bem como a respectiva sequência temporal. O volume de informação a representar faz com que não seja praticável a sua inclusão

neste livro, e por isso optámos por apresentar simplificadamente apenas as actividades e os intervenientes por elas responsáveis.

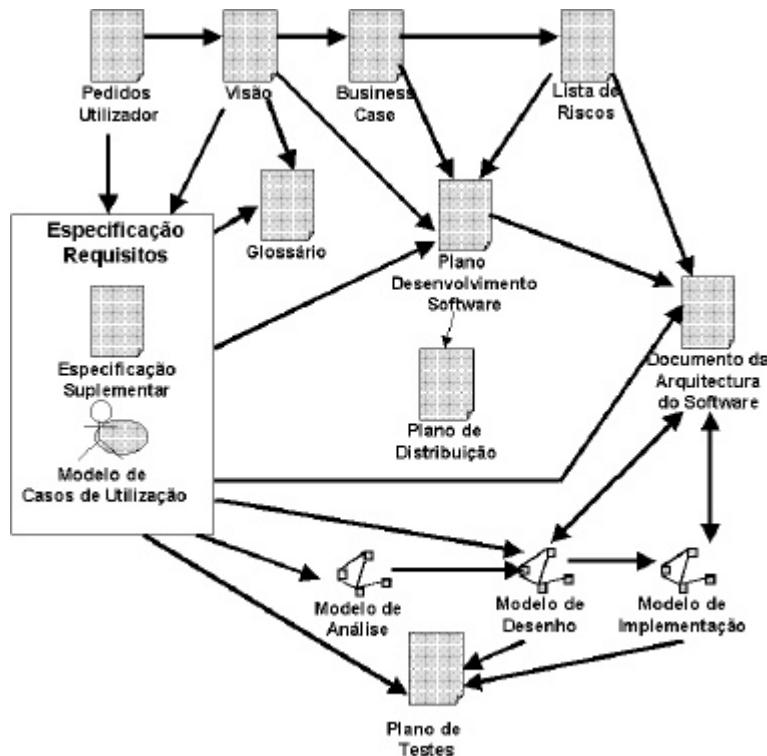



Figura 10.11: Principais artefactos produzidos no âmbito do RUP.

De forma a clarificarmos inicialmente o desenrolar do processo, e a aplicarmos o mesmo ao caso prático, comecemos por abordar o *work-flow* de gestão do projecto.

10.7.1 *Workflow* de Gestão do Projecto

Este *workflow* especifica um conjunto de princípios a aplicar na gestão de projectos de software, sobretudo ao nível do planeamento, alocação de recursos, acompanhamento e controle do projecto, e na identificação e controle dos riscos dos projectos. Não contempla aspectos gerais, que qualquer gestor de projecto deve saber aplicar (não sendo específicos de projectos de desenvolvimento de software), nomeadamente gestão

de recursos humanos, elaboração de orçamentos e gestão de relações entre clientes e fornecedores.

Se as vantagens da aplicação de um processo iterativo são óbvias, o mesmo não acontece com as decisões relacionadas com o número de iterações a realizar, a duração de cada uma, os objectivos a atingir ou a intensidade das actividades de cada *workflow*. A evolução de um projecto, segundo a abordagem preconizada pelo RUP, torna impraticável a elaboração de um plano detalhado e minimamente credível logo no arranque do projecto, uma vez que o nível de conhecimento sobre o problema em análise vai aumentando no decorrer do projecto. Por isso, a estratégia preconizada pelo RUP assenta na existência de dois planos:

- Plano de fases: um plano de alto nível, elaborado inicialmente para as fases, e revisto periodicamente.
- Planos de iterações: um plano por cada iteração elaborado mais próximo do momento da sua execução.

O primeiro inclui as datas dos principais objectivos a atingir (*milestones*) em cada uma das quatro fases, e dos objectivos intermédios que correspondem às datas de fim de cada iteração. Para além destas questões temporais, inclui ainda informação de alto nível sobre a alocação de recursos ao longo do tempo. É um documento de pequena dimensão (2 páginas no máximo), elaborado no início da fase de concepção; pode ser revisto sempre que necessário.

Adicionalmente ao plano de fases deverá ser elaborado um plano por cada iteração, utilizando as notações usuais (diagramas de Pert, Gantt, etc). Normalmente, e em cada momento, deverão estar elaborados dois planos deste tipo, um da iteração actual e outro da imediatamente seguinte, uma vez que para as restantes iterações existirá sempre no futuro informação mais correcta para proceder à respectiva elaboração. Os planos das iterações definem as actividades a um nível mais elementar (concretizando orientações genéricas propostas pelos *workflows* do RUP) e a alocação de recursos correspondente.

Figura 10.12: Actividades do workflow de Gestão de Projecto realizadas pelo Gestor do Projecto.

O principal interveniente neste *workflow* é claramente o gestor do projecto, que é responsável pela elaboração da grande maioria dos artefactos, sendo de destacar os seguintes (ver Figura 10.12):

- *"Business Case"*.
- "Planos de Fases".
- "Planos das Iterações".
- "Plano de Desenvolvimento do Projecto", que inclui o Plano de Aceitação do Produto Final, a Lista de Riscos e o Plano de Gestão dos mesmos, o Plano de Resolução dos Problemas e o Plano de Elaboração de Métricas.

Tal como representado na Figura 10.12, as actividades realizadas pelo gestor de projecto ao longo deste *workflow* podem ser agrupadas em função dos momentos em que ocorrem. Para além do gestor de projecto existe ainda a figura do auditor ou revisor do projecto, que realiza diversas actividades cujo objectivo é garantir a qualidade de cada um dos artefactos produzidos, e cujas actividades estão representadas na Figura 10.13.

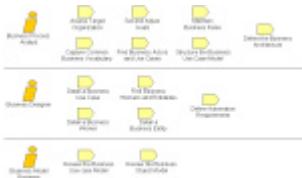



Figura 10.13: Actividades do workflow de Gestão de Projecto realizadas pelo Auditor do Projecto.

A elaboração de planos credíveis está relacionada com a determinação da relação correcta entre recursos, prazos e âmbito. Estes parâmetros do projecto podem ser definidos por comparação com experiências do passado de modo a decidir o esforço e a duração relativa das diversas fases, bem como do número de iterações e duração de cada uma. Estas questões serão discutidas detalhadamente no final deste capítulo com a apresentação de um caso prático (ver Secção 10.8).

10.7.2 Workflow de Modelação do Negócio

As actividades realizadas neste *workflow* têm por objectivo compreender o funcionamento da organização, identificar problemas e oportunidades de melhoria e garantir uma base de entendimento comum entre utilizadores e informáticos, a partir da qual os requisitos do sistema podem ser identificados. Os principais artefactos a produzir são: um documento que reflecte a percepção adquirida sobre o negócio; diagramas de casos de utilização relacionados com negócio; e um modelo dos objectos do negócio.

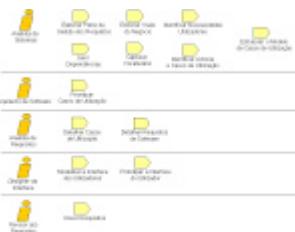
As principais actividades a realizar durante este workflow podem ser observadas na Figura 10.14.

Figura 10.14: Actividades do workflow de Modelação de Processos do Negócio.

As actividades deste *workflow* podem ser realizadas mesmo quando o objectivo não seja o desenvolvimento de software, mas é necessário compreender melhor o negócio de modo a, eventualmente, reformular

alguns dos seus processos. A utilização de técnicas de modelação do negócio (casos de utilização) idênticas às utilizadas para a captura dos requisitos, facilita o mapeamento entre ambos os modelos. Na perspectiva de desenvolvimento de software, este *workflow* é considerado opcional.

10.7.3 Workflow de Requisitos

As actividades realizadas neste *workflow* têm como objectivo principal efectuar a descrição das funcionalidades do sistema e obter o respectivo consenso dos utilizadores. Esta descrição permite delimitar o âmbito do sistema, e constitui a base para a definição dos componentes tecnológicos e para a elaboração das estimativas de custos e esforço para efectuar o seu desenvolvimento. Estes requisitos não são apenas de natureza funcional, isto é, que resultam das necessidades dos utilizadores e dos processos de negócio; outras categorias de requisitos deverão ser identificadas através das actividades realizadas neste *workflow*, nomeadamente:

- Facilidade de utilização do sistema, envolvendo factores como aspectos da interface homem-máquina e elaboração de documentação.
- Fiabilidade: tem a ver com a previsibilidade do funcionamento, gravidade dos erros e precisão do sistema.
- Desempenho: impõe restrições de natureza quantitativa aos requisitos funcionais, podendo reflectir-se em aspectos tecnológicos (por exemplo, duração de uma operação ou memória utilizada).
- Capacidade de suporte: relacionado com o controle de qualidade, a manutenção e o suporte ao sistema após entrada em produção.

O diagrama que, por excelência, deve ser utilizado para modelar as interacções dos utilizadores com o sistema é o diagrama de casos de utilização. Para identificar os requisitos do sistema, são produzidos (ou alterados) diversos artefactos:

- As necessidades e funcionalidades principais são descritas no documento da visão, que fornece uma descrição de alto nível do sistema.

- O detalhe dos requisitos de natureza funcional é representado num modelo de casos de utilização e os restantes são especificados em "Especificações Suplementares".

Para além de outros artefactos, são ainda produzidos o "Glossário" e o "Protótipo da Interface", a validar com os utilizadores. Na produção destes artefactos encontram-se envolvidos analistas de sistemas, especialistas em casos de utilização e especialistas em desenho da interface. As actividades realizadas por estes intervenientes podem ser observadas na Figura 10.15.

Figura 10.15: Actividades do workflow de Requisitos.

10.7.4 Workflow de Análise e Desenho

As actividades deste *workflow* permitem elaborar a especificação técnica do sistema, a partir das especificações funcionais e suplementares produzidas pelo *workflow* anterior. Está, pois, em causa a identificação de uma estratégia e de uma arquitectura para o sistema futuro, que podem ser ajustadas de forma iterativa. Enquanto a análise se concentra na identificação de um conjunto de classes e sistemas que permitem representar os requisitos dos utilizadores (esquecendo por vezes alguns dos requisitos não funcionais), o desenho utiliza o resultado da análise e integra as restrições resultantes dos requisitos não-funcionais, e do ambiente e da arquitectura tecnológica, de forma a garantir a satisfação de todos os requisitos.

Os principais intervenientes neste *workflow* são o arquitecto de software e diversos elementos especializados em tarefas de especificação. São estes intervenientes que estão envolvidos na construção dos principais artefactos do *workflow*, o "Modelo do Desenho" e o "Documento da Arquitectura de Software".

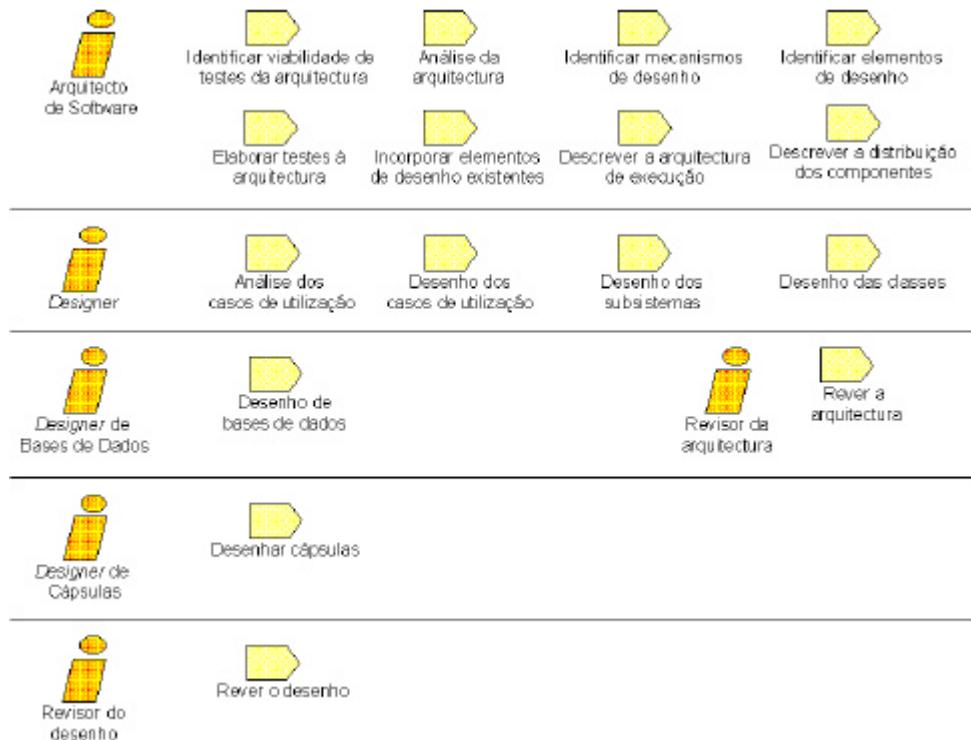



Figura 10.16: Actividades do workflow de Análise e Desenho.

10.7.5 *Workflow de Implementação*

Através deste *workflow* pretende-se definir a estrutura do sistema e implementar os respectivos componentes; testar individualmente cada um; e efectuar a integração dos vários componentes, de modo a construir um sistema executável. São por isso realizadas tarefas associadas à programação e integração, o que torna necessária a participação de intervenientes do tipo programador e integrador de sistemas, com o apoio do arquitecto e do revisor de código. Os principais artefactos a produzir por este *workflow* são os seguintes:

- "Componentes", que são unidades de código, elementares ou que resultam da agregação de outros componentes.

- "Subsistemas de Implementação", constituídos por uma colecção de componentes e outros subsistemas de implementação. O seu conjunto constitui o sistema global.
- "Plano de integração" de uma *release* parcial, que define a ordem de desenvolvimento e de integração de componentes.

As principais actividades realizadas estão representadas no diagrama da Figura 10.17.

Figura 10.17: Actividades do workflow de Implementação.

10.7.6 Workflow de Testes

Este *workflow* garante: (1) a integração entre os componentes implementados; (2) a correcção dos erros detectados antes da instalação do sistema; (3) a conformidade entre o sistema implementado e os requisitos definidos pelos utilizadores.

A preocupação de garantir a qualidade do produto final, através da realização de testes frequentes, leva a que os mesmos possam ser realizados desde muito cedo no desenrolar do projecto. Por exemplo, podem ser efectuados testes aos protótipos desenvolvidos logo na fase de elaboração, o que justifica a existência deste *workflow* nessa fase. As actividades previstas neste *workflow* garantem a realização de diversos tipos de testes, nomeadamente de integração, de sistema e de aceitação, para além de outros relacionados com questões de natureza tecnológica (de carga, de instalação, de desempenho).

Estão envolvidos neste *workflow* dois tipos de intervenientes, responsáveis pelo planeamento e pela execução de testes, e que produzem diversos artefactos, nomeadamente o "Plano de Testes", o "Modelo de Testes" (representação do que será testado e de como será feito) e os "Resultados dos Testes".

As actividades do *workflow* de Testes podem ser observadas na Figura 10.18:

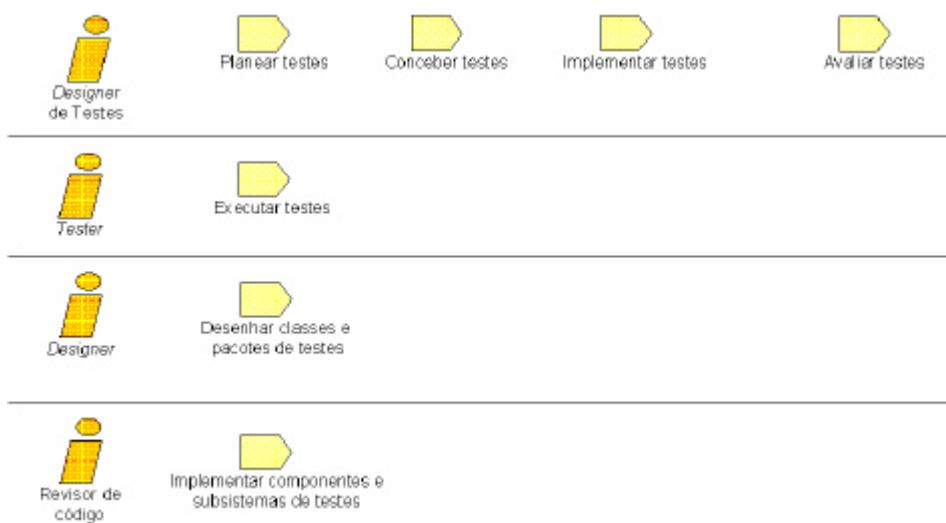



Figura 10.18: Actividades do workflow de Testes.

10.7.7 Workflow de Instalação

De forma a garantir o sucesso do sistema desenvolvido, é essencial efectuar a sua disponibilização para os seus utilizadores finais da forma mais adequada e com o mínimo impacto no seu trabalho. É este o objectivo do *workflow* de instalação. De modo a que o objectivo seja alcançado, está prevista a realização de diversas actividades de apoio, como sejam a produção de manuais de formação e de utilização do sistema. Uma lista mais completa de actividades pode ser observada na Figura 10.19.

Figura 10.19: Actividades do workflow de Instalação.

O nível de abrangência do RUP pode ser observado também neste *workflow*, uma vez que prevê actividades diferentes, consoante a forma como a produção e disponibilização do software é efectuada: através de mecanismos de *download* do software, ou através de produção em massa numa fábrica.

10.7.8 Workflow de Gestão da Configuração e das Alterações

O *workflow* de gestão da configuração e das alterações tem por objectivo garantir e manter a integridade e consistência dos vários artefactos produzidos ao longo do processo. Neste âmbito, podemos estar a falar quer dos requisitos dos utilizadores, quer dos modelos desenvolvidos ou do código propriamente dito.

Os principais intervenientes neste *workflow* são o gestor de configuração e o gestor de controle de alterações, sendo naturalmente envolvidos outros elementos (implementadores, integradores, arquitec-

to, e qualquer outro interveniente que possa solicitar alterações). Os principais artefactos produzidos durante este *workflow* são o "Plano de Gestão de Configuração" e os "Pedidos de Alterações". As actividades previstas são apresentadas na Figura 10.20.

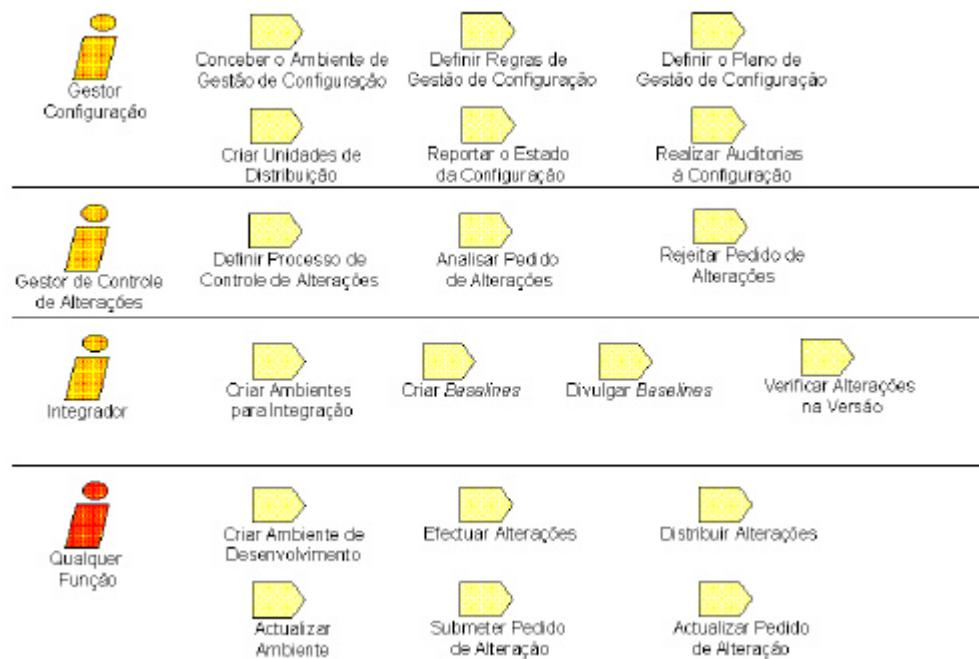



Figura 10.20: Actividades do workflow de Gestão da Configuração e das Alterações.

10.7.9 Workflow de Ambiente

O *workflow* de ambiente tem por objectivo dar apoio aos intervenientes informáticos que estão relacionados com o desenvolvimento do software e com a utilização de ferramentas e aplicação do processo metodológico. Consequentemente, este *workflow* prevê a aplicação de princípios relativos à selecção e aquisição de ferramentas, à configuração, gestão e melhorias a aplicar ao processo.

Como se pode observar na Figura 10.8, as actividades deste *workflow* estão concentradas sobretudo no início do projecto, uma vez que

actividades como selecção de ferramentas e a definição dos princípios metodológicos terão que ser realizadas nesta altura. As suas activida-des são representadas no diagrama da Figura 10.21.

Figura 10.21: Actividades do workflow de Ambiente.

10.8 Enunciado do Caso de Estudo DGD

De forma a demonstrar na prática como podem ser concretizados alguns dos principais conceitos do RUP, vamos introduzir de seguida um caso prático fictício (qualquer semelhança com a realidade é mera coincidência!), e com base nele vamos discutir a aplicação do RUP. Nesta discussão, procuramos sobretudo demonstrar como se planeiam e desenrolam as actividades, os *workflows* e as iterações ao longo de um projecto, sem referir questões de modelação, que foram já apresentadas na Parte 2, e que serão o principal foco do capítulo seguinte, dedicado ao ICONIX.

10.8.1 Enunciado

Suponha-se a existência de um organismo público responsável pela atribuição e gestão de toda a documentação, de natureza oficial e legal, que um cidadão tem. Vamos atribuir a esse organismo a designação de Direcção Geral da Documentação (DGD). Entre a documentação abrangida podemos salientar o bilhete de identidade, o passaporte, o cartão de contribuinte, o cartão de eleitor, a carta de condução, o cartão do serviço nacional de saúde. Toda a informação que se descreve de seguida, se bem que apresentando alguma lógica, não pretende associar-se (ou induzir!) a nenhum organismo em concreto.

A DGD foi criada na sequência da fusão de vários organismos que anteriormente se encontravam na dependência de diversos ministérios, e cuja cultura organizacional de cada um e o ambiente tecnológico dos sistemas de informação respectivos eram muito diversos.

Para além disso, a dispersão geográfica é ainda hoje uma realidade; mesmo na cidade de Lisboa, onde se situavam as sedes de todos os organismos originais, não foi possível até à data efectuar a sua concentração num único edifício. Em localidades do interior do país, existem diversos edifícios da DGD, mas cada um apenas trata de um documento específico.

A integração, que se esperava que trouxesse benefícios óbvios para o cidadão, uma vez que passou a existir apenas um único organismo para

tratar de qualquer assunto relacionado com documentação, não produziu os resultados desejados. Os sistemas de informação originais permanecem separados, e são tecnologicamente muito diversos, desde os ambientes *mainframe* IBM para a gestão das cartas de condução, até aos sistemas Unix para o bilhete de identidade, passando por outro tipo de ambientes (VAX, AS/400, plataformas Windows, etc).

Na altura da criação da DGD, foi lançado um projecto de uniformização dos sistemas (o projecto piloto inicial apenas incluía dois) que falhou, com custos elevados. As questões culturais, a inércia política, o receio da perda de privilégios adquiridos, a falta de empenhamento dos elementos das duas áreas, os problemas tecnológicos e a incapacidade interna de gestão de projecto (tratou-se de um projecto realizado com recursos exclusivamente da DGD) podem ser apresentadas como as principais razões que motivaram este falhanço.

Os recursos humanos existentes na área de informática, apesar de estarem hoje colocados todos no mesmo departamento, continuam afectos aos sistemas onde originalmente trabalhavam. A média de idades dos trabalhadores ultrapassa os 45 anos, a maioria dos quais nunca trabalhou noutras organizações.

É através da DGD que o cidadão se relaciona com o Estado sobre quaisquer assuntos relacionados com documentos. Os principais processos de negócio são os seguintes:

- Registo do pedido de emissão do documento. Envolve a recepção do pedido explicitamente efectuado pelo cidadão ao balcão de uma das "inúmeras" delegações da DGD. Implica apenas receber a documentação que mais tarde é introduzida pelos funcionários da DGD no sistema de informação, sem qualquer controle de prazos desde a entrega até ao registo do processo. O cidadão tem que comprar impressos específicos para cada documento, preenchê-los, e depois de efectuar o pedido, recebe um comprovativo da entrega e nalguns casos efectua um pagamento (por exemplo, no caso do passaporte).

- Validação da possibilidade de emissão do documento. Por exemplo, no caso do pedido do passaporte é verificado o registo criminal do cidadão; no caso do pedido do cartão da segurança social, é contactado o Ministério da Saúde para determinar se a pessoa em causa está ou esteve abrangida por outro sistema de segurança social. Por estas e outras razões que implicam um tratamento diferenciado segundo o tipo do documento, os prazos de entrega não são uniformes.
- Nalguns casos, existe a possibilidade da DGD emitir um documento provisório. É o que acontece no caso do cartão de contribuinte.
- A emissão do documento propriamente dita. Nalguns casos, o documento físico é produzido na própria DGD, noutros casos este é produzido externamente e enviado posteriormente para a DGD. O documento é sempre fotocopiado e a fotocópia armazenada junto do restante processo. Não existem funcionalidades de digitalização implementadas.
- O envio do documento, ou de um aviso para o cidadão, quando o documento estiver pronto, a solicitar ao cidadão o levantamento do documento. No caso em que o documento é logo enviado, é sempre acompanhado por um aviso de recepção.
- A renovação do documento, nos casos em que se aplica (passaporte, bilhete de identidade), e cuja iniciativa parte sempre do cidadão. Não é actualmente feito nenhum controle de documentos fora de prazo, nem dos processos físicos que os suportam. Esta quantidade de papel está armazenada em cinco localizações de grande dimensão, e espalhadas pelos arredores de Lisboa. Quando é necessário esclarecer alguma questão, é preciso requisitar o processo físico, sendo frequentes as situações em que se perdem documentos, ou em que, pelo menos, estes não são localizáveis.
- O registo de averbamentos para alguns documentos. Por exemplo, no caso da carta de condução, a DGD recebe informação da polícia sobre as multas e penas em que um condutor incorreu. Esta informação chega sob a forma de papel, pelo que é necessário efectuar a sua introdução no sistema. O volume que esta informação atinge faz com que exista normalmente um atraso de vários meses.
- Pedidos diversos de informação, que o cidadão efectua à DGD sobre o estado dos seus documentos, ou esclarecimentos sobre procedimentos a realizar.

Para além da informação especificamente relacionada com cada um dos documentos, a DGD recolhe e regista informação sobre o cadastro do cidadão, que lhe chega de diversas fontes (policia, tribunal, ministérios), e que o cidadão pode solicitar através de um pedido de Certidão do Cidadão. Esta informação inclui multas, condenações, louvores, condecorações, medalhas e títulos atribuídos. Uma parte significativa desta informação consta do Diário da República, que não se encontra digitalizado.

Todos estes problemas levaram a tutela governamental a considerar insustentável a manutenção da situação, e em função da aparente incapacidade interna de solucionar os problemas, foi decidido contratar uma empresa externa para implementar um sistema de informação global, integrado, cujo principal objectivo é servir e simplificar a relação com o cidadão. Nesse sentido, um dos pressupostos do projecto era a utilização da Internet através da disponibilização de diversos serviços. Perante este problema de alguma complexidade, a empresa seleccionada resolveu aplicar o RUP ao longo do projecto.

10.9 Resolução do Caso de Estudo DGD

O primeiro grande desafio que se coloca, e que se procura resolver durante a primeira e (normalmente) única iteração da fase de concepção, é a elaboração do artefacto **plano do projecto**. Tal pressupõe a correcta definição do número de ciclos que o projecto terá, da duração do primeiro ciclo, do número de iterações e respectiva duração. Para isso, é inicialmente elaborado um **plano da primeira iteração**, onde são identificadas as actividades a realizar para obter um conhecimento de alto nível do negócio. Por conseguinte, as primeiras actividades a realizar pertencem ao *workflow* de requisitos, e incluem:

- **Elaborar o documento da visão do negócio**. É uma actividade que deve ser realizada conjuntamente pelos analistas de sistemas e pelos utilizadores, sendo produzido o artefacto **visão do negócio**.
- **Identificar as necessidades e as expectativas dos utilizadores**, que deve resultar no artefacto **necessidades dos utilizadores**.

- **Identificar os principais casos de utilização**. Através de reuniões com os utilizadores chave de cada departamento, previamente nomeados, e de questionários que procuravam identificar as principais lacunas dos utilizadores. Como resultado desta actividade será iniciada a produção do **modelo de casos de utilização**.
- **Iniciar a elaboração de um glossário de termos**, de modo a simplificar e manter consistente o modelo de casos de utilização.
- **Elaborar o *business case* e a lista de riscos** (que são também dois artefactos produzidos), como consequência da percepção que entretanto foi obtida das actividades anteriores.

Descrever cada uma das actividades anteriores, bem como o conteúdo dos respectivos artefactos seria demasiado detalhe para incluir neste exemplo. No entanto, mesmo assim gostaríamos de apresentar alguma informação relevante sobre estes artefactos, de modo a transmitir ao leitor a percepção sobre o respectivo conteúdo.

Visão do Negócio

O objectivo deste documento é recolher, analisar e definir as funcionalidades e requisitos de alto nível do sistema.

No caso concreto do problema aqui exposto, ele deveria incluir uma secção onde fossem abordados, mais extensamente, e entre outros, os seguintes tópicos.

Problema: A DGD é um organismo público cujo objectivo é gerir todas as relações entre o Estado e o cidadão, relacionadas com assuntos de documentação. Resultou da fusão de diversos organismos existentes, e cuja integração, quer ao nível do negócio quer dos sistemas de informação nunca foi efectivamente conseguida. Muitos processos de negócio apresentam claras ineficiências. Os sistemas de informação não são adequados para a prestação de serviços de qualidade ao cliente final …

Afectados: Este problema afecta quer o trabalho dos recursos internos da DGD, quer o cidadão que com ela contacta…

Impacto: Custos financeiros não optimizados. Recursos humanos em excesso e mal aproveitados, realizam muito trabalho manual, concentrados em tarefas de menor valor acrescentado. Elevado volume de documentação em papel…

Sistemas actuais: Um conjunto de ambientes tecnologicamente heterogéneo, muitos deles ultrapassados, desenvolvidos internamente, com pouca flexibilidade.

Interessados: Deveriam ser referidas pessoas tais como o Secretário de Estado responsável, o Director Geral, os Chefes dos Departamentos, e caracterizadas as suas posições e interesses no sistema. O outro grande interessado seria sempre o cidadão…

Solução: Sistema de informação integrado, através de diversos canais (balcão, telefone e Internet). Privilegiar critérios de qualidade do atendimento e tempo de resposta. Automatização de tarefas rotineiras, libertação de recursos para outras tarefas de maior valor acrescentado…

Lista de Riscos

A lista de riscos deve enumerar todos os riscos que, potencialmente, possam ocorrer ao longo do projecto. Como exemplo, no caso da DGD poderíamos identificar o risco **Recursos Humanos pouco motivados**.

Ranking do Risco: Muito elevado.

Descrição: Os Recursos Humanos (RH) disponíveis apresentam deficiências graves, mesmo inultrapassáveis, em termos do conhecimento de tecnologia, da aplicação de metodologias de desenvolvimento bem definidas (como é o caso do RUP), e mostram pouca abertura a novas ideias.

Impacto: O fraco envolvimento dos RH actualmente existentes no projecto pode comprometer o sucesso da migração para um novo siste-

ma, caso assumam uma posição reactiva ("eu só dou o que me pedirem").

Indicadores: Para avaliar a existência do risco, poderíamos utilizar grandezas como tempos de atraso no fornecimento de informação pedida.

Estratégias para Evitar Riscos: Introduzir medidas associadas a recompensas financeiras, que promovam a cooperação de todos, por exemplo "se o projecto terminar no prazo previsto todos os funcionários receberão três salários adicionais".

Plano de Contingência: Na impossibilidade de contar com a colaboração dos elementos actualmente existentes para explicar a estrutura de dados, pode justificar-se a introdução de alguma informação à mão, a partir dos processos em papel, ou por técnicas de OCR (*optical character recognition*), uma vez que esta última será sempre necessária no sistema em produção.

Estes riscos não são apenas do negócio, e podem também ser de natureza tecnológica. Por exemplo, na DGD, a utilização de tecnologias Internet, um dos pressupostos da solução, poderia apresentar problemas de desempenho e escalabilidade graves, dado o volume de informação a tratar, e caso a arquitectura não fosse adequadamente definida. Por isso, este risco deveria ser identificado desde o início do projecto, e para avaliar o seu impacto, poderia ser elaborado um pequeno protótipo de uma funcionalidade de consulta de dados de processos de cartas de condução, num ambiente de testes semelhante ao ambiente final de produção. Tal permitiria verificar que estas tecnologias não apresentavam os problemas inicialmente previstos, desde que fossem seguidos um conjunto de princípios, técnicas e boas regras de programação.

Plano das Fases

Com esta informação recolhida (e recordemos que ainda estamos na iteração da fase de concepção), seria chegada a altura de **avaliar se a visão de alto nível obtida pode ser justificada financeiramente**, considerando para isso os investimentos a realizar, as estimativas de recursos, os critérios de sucesso. Entramos assim nas actividades do *workflow* de Gestão de Projectos. Esta avaliação permite detalhar o ***Business Case*** e a **Lista de Riscos**, e elaborar o **Plano das Fases**, que apresenta uma primeira estimativa para as datas de início e fim de cada uma das 4 fases, bem como dos principais pontos de controlo.

Na sequência deste plano, deveria ser possível **obter uma aprovação prévia dos interessados,** relativamente à visão e à viabilidade do projecto. A partir destas são **prioritizados os casos de utilização** e as funcionalidades, e o **plano inicial é detalhado**, sendo definidas o número de iterações para cada fase. Vejamos pois, em termos concretos, como poderia ser efectuado este planeamento no caso da DGD.

A dimensão do projecto da DGD implicou um esforço apreciável, e um número de recursos técnicos e humanos elevado. A partir do conjunto de funcionalidades identificadas, foi aplicado um modelo de estimativa de custos baseado no COCOMO II (para o leitor mais interessado, recomenda-se a leitura de [Bohem00]), e constatou-se que a duração global estimada ultrapassaria os três anos de desenvolvimento, com uma equipa que atingiria no máximo 60 elementos. Estas estimativas permitiram também rever os custos financeiros do projecto.

Dado o prazo global estimado, optou-se por disponibilizar o sistema de forma incremental, completamente funcional em cada versão. Foram definidos três **ciclos,** de acordo com a seguinte lógica:

- **Primeiro ciclo**: Implementação de todas funcionalidades de natureza operacional realizadas pelo pessoal da DGD, com substituição completa de todos os sistemas existentes. A estimativa apontava para um ano e meio de duração.

- **Segundo ciclo**: Implementação de todas funcionalidades relacionadas com a gestão integrada de canais alternativos para o contacto com o cliente, nomeadamente Internet e telefone. A estimativa apontava para um ano de duração.
- **Terceiro ciclo**: Implementação de um sistema de informação de gestão, a partir do qual os quadros da DGD poderiam extrair informação agregada, estimativas, indicadores diversos, e fornecer alguma dessa informação ao cidadão. A estimativa de duração deste ciclo apontava para meio ano.

A decisão sobre as funcionalidades a implementar em cada ciclo foi baseada em critérios de urgência das mesmas e em questões óbvias de precedência: em nenhuma circunstância é possível obter informação de gestão, sem que a informação de natureza operacional esteja disponível.

Plano para o Primeiro Ciclo

Depois destas decisões, o próximo passo é **elaborar um plano para o primeiro ciclo**, o que envolve a definição de:

- Duração e esforço de cada fase.
- Número de iterações em cada fase.
- Duração de cada iteração.

Estas decisões são sempre tomadas com base na informação específica de cada projecto, processada pelo modelo de estimativa de custos, que tem em conta dados históricos. Estes mesmos dados indicam que, em termos médios, o esforço e a duração típicas de cada fase obedecem a um padrão que pode ser observado na Tabela 10.1 [Kruchten00].

Fase	Duração	Esforço
Concepção	10%	5%
Elaboração	30%	20%
Construção	50%	65%
Transição	10%	10%

Tabela 10.1: Padrão da duração e esforço relativos das fases do RUP

No entanto, o RUP prevê a aplicação de alguns factores de correcção consoante se verifiquem ou não determinadas situações, algumas das quais ocorrem neste projecto:

- Dado o tipo de cultura existente na DGD, a diversidade de pessoas (e de interesses) com que era necessário contactar, a determinação do âmbito do projecto seria previsivelmente uma actividade difícil. Nestas circunstâncias, aumentou-se a duração e o esforço relativos da fase de concepção.
- Como na DGD não existe qualquer noção de arquitectura, os riscos do projecto são elevados e os recursos humanos apresentam lacunas importantes ao nível da tecnologia a utilizar, a duração relativa da fase de elaboração foi aumentada.
- Como no processo de migração seria necessário substituir um conjunto elevado de aplicações em ambientes muito distintos, a duração relativa da fase de transição foi também aumentada.

No segundo ciclo, e como os principais problemas (riscos do projecto, definição da arquitectura) já estão resolvidos, a situação é precisamente ao contrário, isto é, a duração e o esforço relativos das fases de com-cepção e de elaboração deve ser reduzida.

Assim, para o primeiro ciclo deste projecto utilizar-se-iam as seguintes grandezas:

Fase	Duração Relativa	Esforço Relativo	Duração Absoluta (em dias)
Concepção	15%	10%	54
Elaboração	35%	30%	126
Construção	35%	45%	126

Transição	15%	15%	54

Tabela 10.2: Proposta da duração e esforço relativos das fases do primeiro ciclo do RUP no caso da DGD.

Duração e Número de Iterações

A recomendação do RUP aponta para que a duração de uma iteração se situe entre 2 a 6 semanas, mas é óbvio que tal depende de cada projecto. Os dados históricos permitem afirmar que este valor depende essencialmente do número de pessoas envolvidas, do número de linhas de código a produzir, da familiaridade de toda a equipa com o processo de desenvolvimento iterativo, do nível de automatização. Kruchten disponibiliza indicadores de estimativa da duração de uma iteração, em função quer do número de linhas de código a desenvolver, quer do número de pessoas [Kruchten00].

Linhas de Código	**Número de Pessoas**	**Duração de uma iteração**
5000	4	2 semanas
20000	10	1 mês
100000	40	3 meses
1000000	150	8 meses

Tabela 10.3: Duração de uma iteração em função de outros factores.

Se bem que estes dados não possam ser aplicados de forma indiscriminada, podem ser considerados uma boa referência. No caso da DGD, foram utilizados para identificar a duração de cada iteração (Tabela 10.4).

Fase	**Duração das Iterações**	**Número de Iterações**
Concepção	3 meses	1

Elaboração	3 meses	2
Construção	2 meses	3
Transição	5 semanas	2

Tabela 10.4: Duração e número de iterações no caso da DGD.

Com estes dados a previsão relativamente ao número de iterações tornou-se relativamente fácil:

- Uma iteração preliminar, para a fase de concepção, que não produziria nenhum produto final funcional, mas em que a preocupação principal era a identificação e resolução dos principais riscos do projecto. Dada a sua dimensão, era de esperar que a resolução de alguns destes riscos pudesse implicar a tomada de decisões importantes, a nível político, e daí a duração da iteração. A obtenção de uma visão detalhada, para além de permitir identificar os riscos do projecto, possibilitaria ainda a elaboração de protótipos pontuais para avaliação de alguns riscos (de natureza tecnológica). No entanto, neste momento do processo, é natural que muitos dos riscos sejam de natureza humana e de negócio, e menos de natureza técnica.
- Na fase de elaboração, e uma vez que não existe qualquer noção de arquitectura, e tendo em conta a falta de preparação do tipo de recursos envolvidos, foi decidida a realização de duas iterações. Esta estratégia tem como vantagem poder elaborar um protótipo funcional, que elimina parte significativa dos riscos logo na primeira iteração desta fase, e cujos erros de arquitectura poderão ser corrigidos durante a segunda. Além disso, o contacto com uma interface visual do sistema, se bem que ainda de reduzido detalhe, durante a primeira iteração, ajudará os utilizadores a especificar melhor o sistema durante a segunda iteração.
- Na fase de construção, tal como é apresentado por omissão pelo RUP, estão previstas três iterações, as duas primeiras essencialmente concentradas na implementação dos componentes do sistema, e a terceira mais direccionada para os testes e integração do mesmo.
- Na fase de transição, e dada a complexidade da migração, estão previstas duas iterações, a última das quais quase exclusivamente concentrada em actividades de migração do sistema.

Como resultado desta iteração, obtém-se uma **visão inicial do âmbito do projecto**, o ***business case*** do mesmo e um **plano do projecto** (plano das fases). Uma das principais novidades do RUP em relação à maioria das metodologias que se aplicam a todo o processo de desenvolvimento é que o RUP reconhece que as estimativas iniciais do plano do projecto são aproximadas e devem ser continuamente refinadas.

Primeira Iteração da Fase de Elaboração

No início desta iteração, deve ser **elaborado o respectivo plano**, actividade que como vimos pertence ao *workflow* de Gestão de Projectos. A elaboração do plano de qualquer iteração deve seguir os seguintes passos:

- Identificar critérios objectivos para o sucesso da iteração.
- Identificar os artefactos a produzir ou alterar de forma a garantir o sucesso.
- Identificar as actividades do RUP que devem ser realizadas para produzir os artefactos.
- Utilizar estimativas para identificar a duração e o esforço de cada actividade.

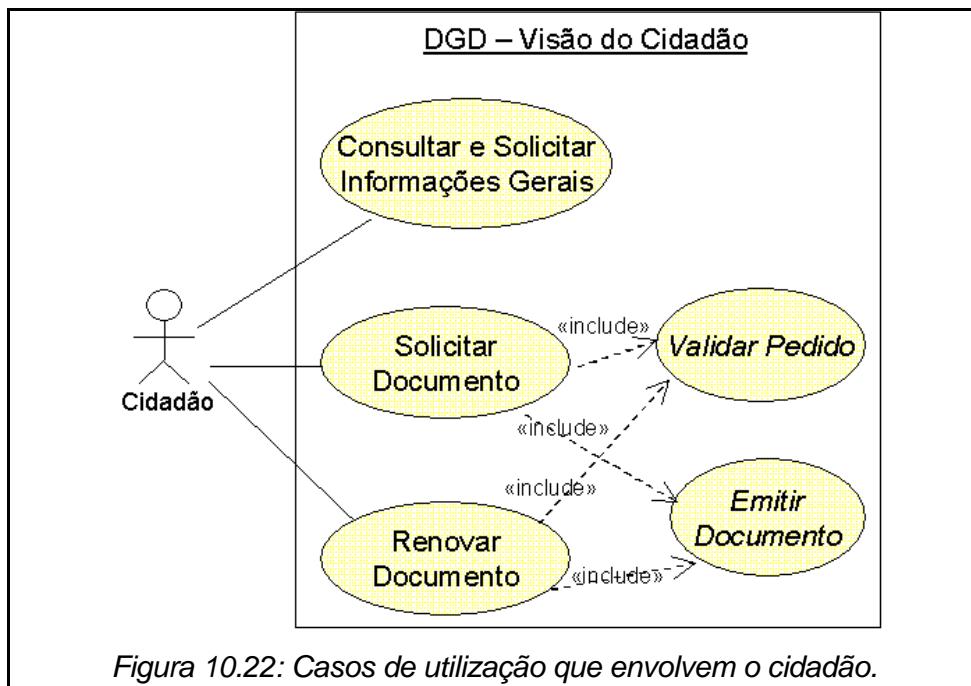
Nas iterações da fase de elaboração, os principais factores a ter em conta para o respectivo sucesso são riscos, cobertura de requisitos e criticidade. Os riscos assumem um papel fundamental, pelo que deverão estar previstas actividades durante esta iteração para os eliminar.

Por exemplo, no caso da DGD a disponibilização de documentos digitalizados, através de tecnologias Internet, poderia apresentar problemas de integração graves, caso não fosse possível integrar o sistema de gestão operacional e as tecnologias de gestão documental. Por isso, foi previsto um cenário para verificar esta integração, através do desenvolvimento de um protótipo.

A criticidade significa que deverão ser consideradas actividades para detalhar as principais funcionalidades do sistema.

No caso da DGD, elas incluiriam todos os serviços que esta presta directamente ao cliente (pedidos de documentos, entrega de documentos, serviços de esclarecimento) e todas as actividades internas que são necessárias para o suporte adequado desse serviço. No entanto, os processos contabilísticos e de gestão de recursos humanos poderiam ser deixados para a segunda iteração da fase de elaboração.

A estratégia de análise dos requisitos do sistema deve abordar primeiro os problemas mais genéricos, deixando para iterações subsequentes decisões que envolvem demasiado detalhe.

Tendo estas orientações em conta, e relativamente à DGD, alguns objectivos poderiam ser definidos para esta iteração:

- Conceber o sistema de forma a permitir que mais de 1000 funcionários em todo o país possam estar simultaneamente no sistema a registar pedidos de documentos, e que outros 2000 possam estar a consultar informação sobre processos existentes. Este cenário implicaria alguns testes de natureza tecnológica, que envolveriam a realização de testes de volume de carga e tempos de resposta para avaliar os aspectos de desempenho, disponibilidade e escalabilidade do sistema.
- Conceber o sistema de modo a eliminar a separação actualmente existente entre a recepção dos documentos ao balcão e a respectiva introdução no sistema.

- Prever uma arquitectura que permita aos utilizadores terem um nível de desempenho aplicacional idêntico (variação máxima entre 10% em relação aos níveis médios), independentemente da sua localização no país. Este cenário implicaria a realização de testes ao nível da infra-estrutura tecnológica de comunicações.
- Identificar e modelar todos os casos de utilização que envolvam o actor cidadão. Apesar de termos colocado este objectivo em último lugar, ele é o mais óbvio de todos, uma vez que o objectivo da fase de elaboração é precisamente a identificação dos requisitos do sistema, e era pressuposto da primeira iteração concentrar a sua atenção em todos os processos que envolvessem o cidadão.

A definição destes objectivos permite identificar as actividades a realizar e elaborar o **Plano da Iteração**. A informação recolhida permite detalhar ainda mais a **Lista de Riscos**, e rever o Plano das Fases iniciado durante a fase de Elaboração.

De seguida, é tomada uma decisão sobre quais os **casos de utilização relevantes** e que condicionarão o desenvolvimento da arquitectura (actividade do *workflow* de requisitos).

No caso da DGD serão, claramente aqueles que estão relacionados com a gestão das interacções com o cidadão, como resumidamente se apresenta na Figura 10.22.

Figura 10.22: Casos de utilização que envolvem o cidadão.

Estes casos de utilização deverão ser descritos em detalhe, e a informação obtida deve ser reflectida no **Modelo de Casos de Utilização**, no caso de se tratarem de requisitos funcionais, ou no documento **Especificação Suplementar**, caso sejam de outra natureza.

No caso da DGD poderemos identificar outros requisitos que não são de natureza funcional:

- O número de utilizadores a introduzir informação no sistema pode ser superior a 1000.
- O número de utilizadores a consultar informação no sistema pode ser superior a 2000.
- O tempo de resposta a qualquer consulta não pode ser superior a 10 segundos.

Com esta informação adicional, é revisto o **Plano da Iteração** e a **Lista de Riscos**, se assim se justificar. A partir dos casos de utilização já especificados, é iniciada a concepção de um **Protótipo da Interface do Sistema**, que é apresentado aos utilizadores para recolher os seus comentários.

No caso da DGD a interface a desenvolver nesta iteração poderia incluir:

- Um ecrã de registo de pedidos de documentos, com dados comuns a qualquer documento. A partir deste ecrã seriam apresentados outros específicos para cada documento.
- Um ecrã de registo de pedidos de informação dos cidadãos.
- Um ecrã de consulta de informação sobre um cidadão, com diversas possibilidades de pesquisa.
- Um ecrã para registar os diversos pagamentos de taxas que o cidadão pode efectuar.
- ….

De seguida são iniciadas actividades do *workflow* de Análise e Desenho, nomeadamente **identificar classes óbvias**, **efectuar prototipagem inicial dos subsistemas** e **analisar casos de utilização em detalhe**. Estas actividades levam à elaboração do **Documento da Arquitectura do Software** e à identificação das classes e objectos correspondentes aos casos de utilização desta iteração. Estas classes são posteriormente refinadas, através da identificação das suas responsabilidades, atributos e relações. As classes consideradas do ponto de vista da arquitectura como relevantes são incluídas no **Documento da Arquitectura do Software**. Finalmente é criada a hierarquia de conceitos associados às classes: estas são atribuídas a pacotes e estes a subsistemas.

No caso da DGD as classes mais importantes e que imediatamente podem ser identificadas como condicionantes de toda a arquitectura serão as de Cidadão, Documento (e suas subclasses) e Operações.

As actividades do *workflow* de Análise e Desenho continuam com o **refinamento da arquitectura**, que é ajustada ao ambiente de implementação seleccionado (sistemas operativos, bases de dados, servidores aplicacionais, etc). Através de diagramas de interacção é modelada a forma como as classes são instanciadas em objectos, pela descrição de como os casos se utilização ou cenários seleccionados são realizados em termos de objectos que interagem entre si. Os requisitos de desenho de cada objecto são reflectidos nos diagramas de interacção, que por sua vez condicionarão a definição das interfaces das classes, havendo pois a necessidade de rever a visão lógica do sistema.

Finalmente, são pela primeira vez analisadas as questões relacionadas com a **distribuição dos componentes da arquitectura**, com o **processamento e com a infra-estrutura tecnológica**. De modo a garantir a consistência de todo o modelo, a arquitectura é revista.

As actividades seguintes pertencem ao *workflow* de Implementação:

- Análise da implementação da arquitectura em termos físicos, definindo a visão de implementação.
- Planeamento da sequência da implementação dos subsistemas, de modo a que possam ser integrados num protótipo da arquitectura.

De seguida é efectuado o planeamento dos testes de integração e de sistema, que são actividades previstas no *workflow* de Testes, sendo produzido o **Plano de Testes**. São também descritos os testes a efectuar, quer em termos de valores quer de procedimentos.

Procede-se à implementação das classes e à respectiva integração (*workflow* de Implementação), o que é efectuado de forma incremental. A arquitectura final, correspondente ao sistema final previsto nesta iteração (que, recorde-se, é um sistema funcional mas incompleto), é

testada (*workflow* de Testes) e os resultados são analisados de forma a garantir que os objectivos dos testes foram atingidos. O protótipo é apresentado aos utilizadores para recolha de comentários e de sugestões.

A iteração termina com a avaliação dos resultados obtidos, nomeadamente a comparação do trabalho realizado com o previsto (em termos de prazos e custos). Em função do trabalho identificado como sendo necessário realizar, pode-se planear mais adequadamente a iteração seguinte. A **Lista de Riscos** e o **Plano de Projecto** poderão ser revistos.

Primeira Iteração da Fase de Construção

Para a fase de construção, os principais objectivos deixam de ser condicionados pelos riscos do projecto, mas sobretudo pela garantia de implementação dos casos de utilização. O planeamento deve reflectir a respectiva criticidade, devendo ser implementados em primeiro lugar os casos de utilização mais críticos, de modo a possibilitar a sua verificação mais cedo e em várias iterações da fase de construção.

No caso da DGD, para a primeira iteração da fase de construção deveriam ser seleccionados os seguintes casos de utilização:

- Entrega de documentos pelo cidadão.
- Pedido de informação do cidadão.

Poderíamos (e, num caso concreto, deveríamos) continuar com a análise das restantes iterações. No entanto, quer por questões de espaço, quer pelo detalhe em que cada vez mais nos envolvemos à medida que avançamos no processo RUP, com a necessidade de definir várias questões de natureza tecnológica, optámos por terminar o exemplo neste ponto. Esperemos que em projectos da vida real o leitor não faça o mesmo!

10.10 Conclusão

Neste capítulo apresentou-se uma panorâmica geral sobre o Rational Unified Process, RUP, que dada a sua dimensão e complexidade foi necessariamente incompleta. No entanto, esta descrição transmite ao leitor as características e conceitos básicos da metodologia, a forma como está organizada e como pode ser aplicada.

Apesar da quantidade de conceitos, procurámos demonstrar com um exemplo credível, como o RUP pode ser posto em prática, dando relevo às actividades realizadas e aos artefactos produzidos. Este exemplo apresentava um grau de complexidade elevada, não só pela dimensão do esforço a realizar, como também devido a todos os problemas de natureza não técnica que lhe estavam associados. É precisamente neste tipo de situações que a abrangência do RUP se revela particularmente útil, uma vez que considera actividades para tratamento da maioria destes problemas.

A complexidade do RUP não deve "assustar" o leitor e desmotivá-lo na adopção do mesmo como metodologia de desenvolvimento a aplicar de forma sistemática. Como dissemos no início deste capítulo, o RUP é um *framework* metodológico muito completo, que pode e deve ser adaptado ou instanciado à realidade concreta de diferentes organizações e tipos de projecto. Os próprios proponentes do RUP assim o entendem, para além de outros autores, tais como Martin Fowler [Fowler00] e Gary Evans [Evans01].

O trabalho futuro será naturalmente a definição e proposta de instâncias do RUP, ou seja, de metodologias baseadas no RUP, mas mais simples, mais especializadas e, por conseguinte, melhor adaptadas a distintas organizações e a diferentes tipos de projectos. É, na nossa opinião, perfeitamente aceitável o aparecimento, a curto e/ou médio prazo, de processos "baseados no RUP", que possam "competir" favoravelmente com o conjunto de metodologias *lightweight* que estão a ganhar adeptos junto da comunidade informática, entre as quais se destaca o ICONIX, que é apresentado no capítulo seguinte.

10.11 Exercícios

Ex.71. Tendo em conta os seus conhecimentos sobre o RUP, quais são os tipos de empresas que maior partido poderão tirar desta metodologia de desenvolvimento? Justifique a sua afirmação

Ex.72. Complete o diagrama de casos de utilização apresentado na Figura 10.22, relativamente ao actor `Cidadão`.

Ex.73. Relativamente à metodologia de desenvolvimento RUP, aplicam-se normalmente critérios uniformes para definir o tempo de duração da fase de concepção, em função de experiências do passado, e introduzem-se alguns factores de ajustamento que podem aumentar ou diminuir esse tempo. Para além dos factores referidos no livro, indique outros que na sua opinião poderão influenciar os valores das referidas estimativas.

Ex.74. Identifique quais os principais riscos que o projecto poderá apresentar, e proponha algumas medidas de correcção.

Ex.75. Elabore o diagrama de classes simplificado referente ao caso de estudo apresentado.

Ex.76. Imagine o desenvolvimento de um sistema de informação de complexidade relativamente baixa, cuja duração total não excederia três meses. Seria possível (e desejável) aplicar o RUP? Em caso afirmativo, que alterações (ou adaptações) introduziria?

Ex.77. O RUP refere que a aplicação do workflow de modelação do negócio é opcional. Em que circunstâncias é que recomendaria a sua inclusão num projecto de desenvolvimento de software.

Capítulo 11 - METODOLOGIA ICONIX

Tópicos

- Introdução
- Visão Geral
- Avisos da Metodologia Iconix
- Enunciado do Caso de Estudo WebDEI
- Resolução do Caso de Estudo WebDEI
- Conclusão

11.1 Introdução

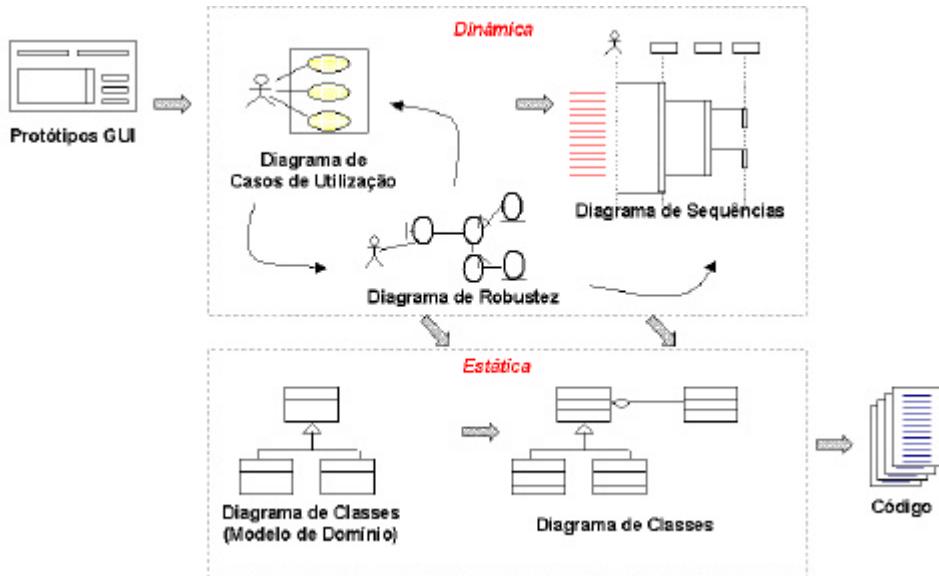
O ICONIX é uma metodologia de desenvolvimento de software promovido pela empresa ICONIX Software Engineering, cujo foco de negócio reside na formação e produção de material para educação em tecnologias de objectos, em particular em CORBA, COM, Java e UML. O principal evangelista do ICONIX é Doug Rosenberg, autor do livro "*Use Case Driven Object Modeling with UML: A Practical Approach*" [Rosenberg99] com base no qual este capítulo foi desenvolvido. Está prevista a edição de um novo livro dos mesmos autores, intitulado "*Applied Use Case Driven Object Modeling*", para o princípio do segundo trimestre de 2001. (Mais informações sobre o ICONIX e respectivos serviços e produtos podem ser obtidas em *www.iconixsw.com*)

O ICONIX define-se como um "processo" de desenvolvimento de software prático, algures entre a complexidade e abrangência do RUP (*Rational Unified Process*) e a simplicidade e o pragmatismo do XP (*Extreme Programming*).

O ICONIX é conduzido por casos de utilização, iterativo e incremental, tal como o RUP, mas sem a complexidade que este preconiza. Por outro lado, é relativamente pequeno e simples, tal como o XP, mas sem eliminar as tarefas de análise e de desenho que aquele não contempla. Por fim, o ICONIX usa o UML como linguagem de modelação e apresenta um alto grau de rastreabilidade[1] (*traceability*).

Discutimos neste capítulo os principais aspectos do ICONIX, primeiro segundo a apresentação da sua visão geral, e seguidamente através da sua aplicação na modelação do caso de estudo WebDEI. O WebDEI consiste num sistema de informação Web a quatro camadas, desenvolvido exclusivamente com tecnologia Java.

11.2 Visão Geral

O ICONIX é uma metodologia de desenvolvimento de software, englobando as seguintes tarefas principais: (1) análise de requisitos; (2) análise e desenho preliminar; (3) desenho; e (4) implementação, conforme se introduz sumariamente de seguida.

A metodologia consiste na produção de um conjunto de artefactos que retratam as visões dinâmica e estática de um sistema, e que vão sendo desenvolvidos incrementalmente e em paralelo, os modelos da estática e os modelos da dinâmica.

A Figura 11.1 ilustra a visão geral do ICONIX. Esta figura revela um aspecto importante da utilização do UML: o facto da implementação de um sistema apenas depender da versão detalhada do diagrama de

---

[1] Rastreabilidade é a capacidade de se seguir as relações entre os diferentes artefactos produzidos no processo de desenvolvimento de software, de forma a poder-se averiguar, por exemplo, o impacto que uma determinada alteração de um requisito tem em todos os restantes artefactos (modelos de análise, desenho, e mesmo de implementação).

classes final. (Parece óbvio, mas muitos indivíduos pensam ainda que se poderiam usar, por exemplo, diagramas de sequência para gerar código automaticamente!)

Figura 11.1: Visão geral do ICONIX.

11.2.1 Análise de Requisitos

A tarefa de análise de requisitos consiste na realização das seguintes actividades (ver Figura 11.2):

- Identificar os objectos do mundo real e todas as relações de generalização, associação e agregação entre esses objectos. Desenhar o correspondente diagrama de classes de alto nível, designado por modelo de domínio.
- Se for razoável (e.g., se for pertinente ou se houver orçamento para tal actividade), desenvolver protótipos de interface homem-máquina (GUI), diagramas de navegação, etc., de forma que os utilizadores e clientes possam entender melhor o sistema pretendido.
- Identificar os casos de utilização envolvidos no sistema. Desenhar os diagramas de casos de utilização realçando os actores envolvidos e as suas relações.

- Organizar em grupos os casos de utilização. Capturar essa organização através de diagramas de pacotes (*packages*).
- Associar requisitos funcionais aos casos de utilização e aos objectos do domínio.

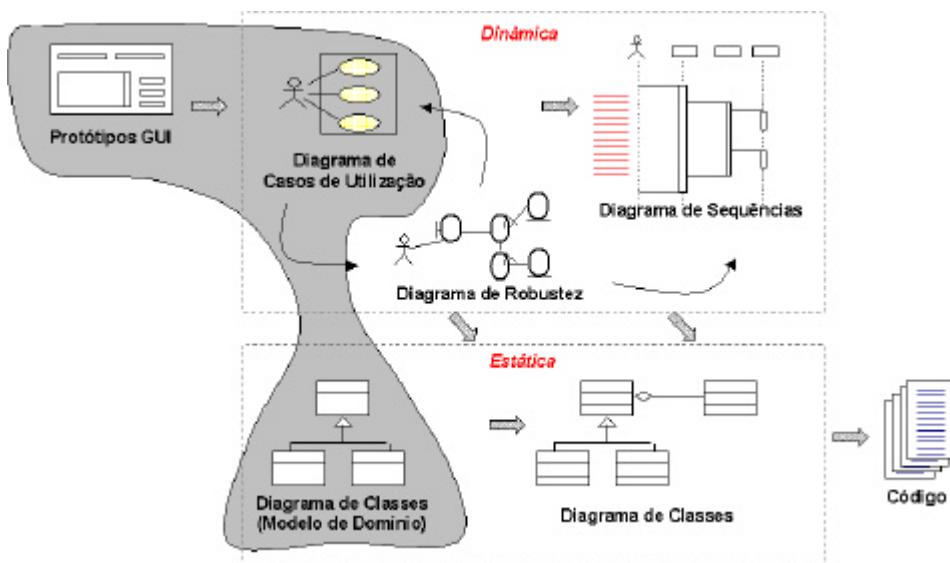
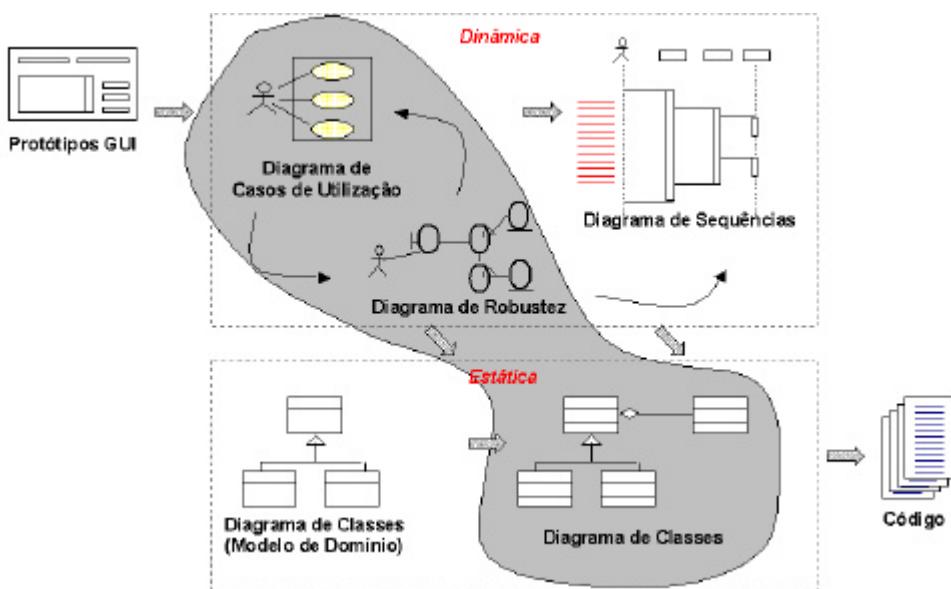



Figura 11.2: ICONIX– Actividades da tarefa de análise de requisitos.

Observação: O ICONIX distingue explicitamente um requisito de um caso de utilização. Em particular, o seu autor distingue-os da seguinte forma:

- Um caso de utilização descreve uma unidade de comportamento.
- Um requisito descreve uma regra que governa o comportamento.
- Um caso de utilização satisfaz um ou mais requisitos funcionais.
- Um requisito funcional pode ser satisfeito por um ou mais casos de utilização.

Ou seja, segundo o ICONIX, há uma relação de muitos-para-muitos entre casos de utilização e requisitos, pelo que tem sentido a última actividade desta tarefa, de associação entre estes dois conceitos. Note-se que existem outros autores que consideram, por outro lado, os casos de utilização o mecanismo ideal para se especificarem os próprios requisitos de um sistema. (Esta relação entre casos de utilização e requisitos é um assunto em discussão na comunidade OO, não existindo até ao momento qualquer consenso.)

11.2.2 Análise e Desenho Preliminar

A tarefa de análise e desenho preliminar consiste na realização das seguintes actividades (ver Figura 11.3):

- Fazer as descrições dos casos de utilização com os cenários principais, cenários alternativos e cenários de excepções.
- Fazer a análise de robustez. Para cada caso de utilização:
  - Identificar um primeiro conjunto de objectos. Usar os estereótipos de classes definidos no perfil “Processos de Desenvolvimento de Software” especificado no UML 1.3 (*boundary*, *control*, e *entity*).
  - Actualizar o diagrama de classes do modelo do domínio, com os novos objectos e atributos entretanto descobertos.
- Terminar a actualização do diagrama de classes de modo a reflectir a conclusão da fase de análise.

Figura 11.3: ICONIX- Actividades da tarefa de análise e desenho preliminar.

11.2.3 Desenho

A tarefa de desenho consiste na realização das seguintes actividades (ver Figura 11.4):

- Especificar o comportamento. Para cada caso de utilização:
  - Identificar os objectos, as mensagens trocadas entre objectos e os métodos associados que são invocados. Desenhar um diagrama de sequência com o texto do caso de utilização do lado esquerdo, e informação do desenho do lado direito. Continuar a actualizar o diagrama de classes com os objectos e atributos entretanto descobertos.
  - Se for relevante, usar diagrama de colaboração para ilustrar as transacções principais entre objectos.
- Terminar o modelo estático, adicionando informação detalhada sobre o desenho (e.g., visibilidade e padrões de desenho)
- Verificar que o desenho satisfaz todos os requisitos identificados.

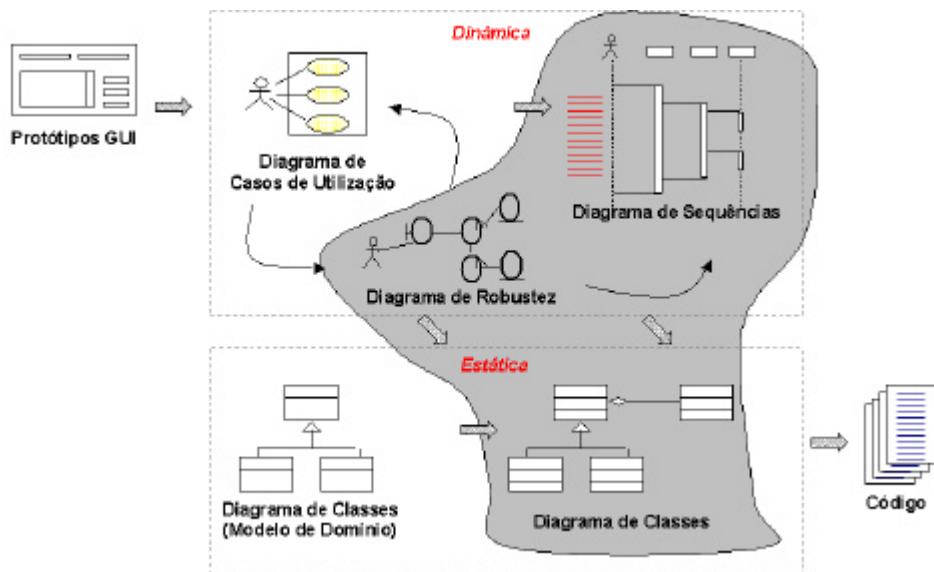



Figura 11.4: ICONIX– Actividades da tarefa de desenho.

11.2.4 Implementação

A tarefa de implementação consiste na realização das seguintes actividades (ver Figura 11.5):

- Consoante as necessidades, produzir diagramas de arquitectura, tais como, diagramas de instalação e de componentes, que apoiem a tarefa de implementação.
- Escrever e, eventualmente, gerar o código.
- Realizar testes unitários e de integração.
- Realizar testes de sistema e de aceitação do utilizador.

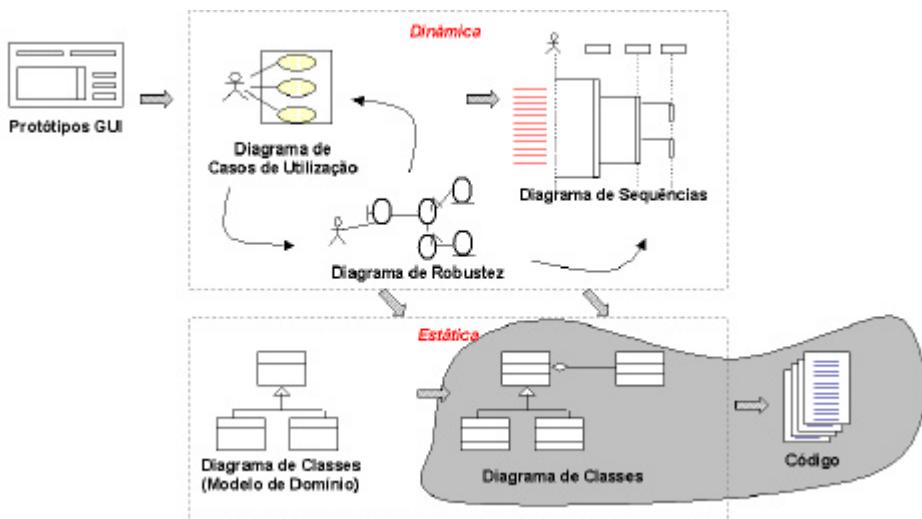



Figura 11.5: ICONIX– Actividades da tarefa de implementação.

11.3 Avisos do Processo ICONIX

O ICONIX lança um conjunto de avisos relativamente a problemas e dúvidas comuns, que deverão ser tidos em conta pelos intervenientes do processo.

Os avisos relacionados com a realização da tarefa de análise de requisitos têm como principal objectivo evitar a perda de tempo com detalhes desnecessários. Designadamente:

- Na produção dos modelos de domínio:
  - Não perder demasiado tempo com a inspecção gramatical.
  - Não endereçar o desenho da multiplicidade demasiado cedo no projecto.
  - Endereçar a agregação e composição apenas na fase do desenho detalhado.
- Na produção dos modelos de casos de utilização:
  - Não começar a escrever os casos de utilização até se conhecer bem como é que os utilizadores irão actuar.
  - Não passar semanas a construir modelos de casos de utilização elaborados e bem desenhados, mas a partir dos quais não é possível construir-se um adequado desenho de classes.

- Não perder muito tempo em discussões sobre quando e onde usar relações "*include*" ou "*extend*".
- Não usar *templates* textuais de casos de utilização muito longos ou complexos.

Avisos a ter em conta na realização da tarefa de análise e desenho preliminar:

- Não procurar fazer desenho detalhado nos diagramas de robustez
- Não perder tempo a tentar aperfeiçoar os diagramas de robustez à medida que o desenho evolui.

Avisos a ter em conta na realização da tarefa de desenho:

- Na produção dos diagramas de sequência:
  - Não procurar associar comportamento aos objectos antes de se ter um ideia concreta sobre o seu significado e interesse para o sistema.
  - Não começar a desenhar um diagrama de sequência antes de se ter completado o diagrama de robustez correspondente.
  - Não focar a atenção (e esforço) na definição de métodos "*get*" e "*set*" em detrimento dos métodos reais. (Estes métodos de acesso e alteração dos atributos podem ser facilmente gerados automaticamente a partir de uma ferramenta CASE.)
- Na produção dos diagramas de estado:
  - Não desenhar diagramas de estados para objectos com apenas dois estados.
  - Não modelar o que não é necessário modelar.
  - Não desenhar diagramas de estados só porque se consegue desenhá-los.

11.4 Enunciado do Caso de Estudo WebDEI

O caso de estudo "WebDEI" consiste num sistema de informação Web para gerir alguns processos de negócio de um departamento de engenharia informática de uma instituição de ensino superior, que passaremos a designar apenas por "DEI", e o respectivo sistema de

informação por "WebDEI". Considere-se que tanto o DEI como o WebDEI são realidades fictícias e que o WebDEI aqui apresentado é, por motivos didácticos, uma natural simplificação da realidade.

Descreve-se de seguida, de forma propositadamente pouco estruturada, os requisitos e arquitectura do WebDEI, assim como o modelo de dados e a funcionalidade pretendida.

Pede-se que o leitor desenvolva o referido sistema por aplicação do processo ICONIX. (Ou que pelo menos reflicta sobre o seu processo de desenvolvimento.) As tarefas mais importantes contempladas neste livro referem-se à fase de concepção, em particular às tarefas de identificação de requisitos, de análise e de desenho.

11.4.1 Introdução

O DEI tem por missão promover, organizar e realizar actividades de ensino (licenciaturas, mestrados, doutoramentos, pós-graduações, cursos e/ou seminários de curta duração, de carácter profissionalizante), de investigação e de desenvolvimento, bem como actividades de comsultoria ligadas ao tecido social e económico.

Integram o DEI um conjunto de membros docentes, os quais pertencem a áreas científicas. Cada área científica é coordenada por um professor e é constituída por vários docentes (professores e assistentes). Adicionalmente, a área científica é atribuído um conjunto de disciplinas, pelo qual é responsável, que podem ser leccionadas em diferentes licenciaturas ou outros tipos de cursos.

O DEI é ainda responsável pela promoção e organização de diversos eventos de carácter científico e profissional bem como por promover mecanismos de disseminação automática de informação.

11.4.2 Arquitectura Geral

Nesta secção são descritos os aspectos básicos da arquitectura do WebDEI. As questões em aberto deverão ser alvo de reflexão do leitor de modo a tomar as suas decisões fundamentadamente. O sistema

deve funcionar num ambiente aberto e distribuído, como a Internet, de acordo com a arquitectura geral conforme ilustrado na Figura 11.6.

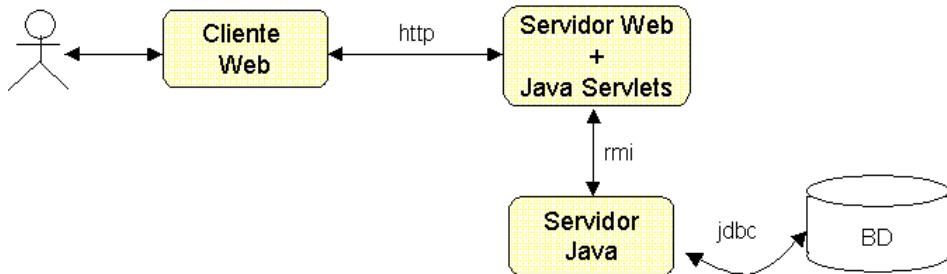



Figura 11.6: Arquitectura geral do WebDEI.

O WebDEI apresenta uma arquitectura a quatro camadas, a primeira das quais é caracterizada pelo cliente Web, com capacidades de apresentação de páginas HTML; a segunda camada consiste em componentes *servlets* Java (executadas no contexto de um servidor Web) que produzem páginas HTML dinamicamente; a terceira camada consiste no servidor aplicacional Java, que mantém a lógica dos processos e medeia o acesso à base de dados; e por fim, a quarta camada consiste num sistema de gestão de base de dados, responsável pela operação da base de dados envolvida.

Assume-se que todas as comunicações entre os utilizadores e a máquina do servidor são efectuadas via HTTP; que a comunicação entre os *servlets* e o servidor Java se realiza via Java RMI (*Remote Method Invocation*); e que a comunicação entre o servidor Java e o gestor da base de dados se realiza via JDBC (*Java Data Base Connectiviy*).

Utilizadores/Actores

O WebDEI deve suportar os seguintes tipos de utilizadores:

- Utilizador anónimo: Conjunto de utilizadores não registado no sistema, i.e., público em geral. Estes utilizadores apenas têm acesso a informação que lhes é transmitida através de documentos HTML. Estes documentos são mantidos e/ou gerados por um programa associado ao servidor Web. Tipicamente, os documentos gerados deverão corresponder a informação que não seja crítica e confidencial como por exemplo: consulta de informação sobre áreas científicas, sobre docentes, sobre disciplinas, etc.
- Docente: Utilizadores que acedem ao sistema através de páginas HTML geradas dinamicamente (pelos *servlets*). Podem configurar a sua página pessoal através de um formulário que lhes é apresentado. Podem registar eventos/notícias. Podem receber informação periodicamente.
- Gestor: Utilizadores responsáveis pela gestão e configuração da informação do sistema. Estes acedem ao sistema localmente. Tipicamente, os gestores são responsáveis pela gestão (assume-se as operações usuais de introdução, remoção e alteração) de utilizadores, áreas científicas, disciplinas, docentes, etc. São ainda responsáveis pela gestão de todos os parâmetros de configuração envolvidos (e.g., definição da periodicidade do envio automático de informação de eventos/notícias).

11.4.3 Tipos Básicos de Informação (Modelo de Dados)

A base de dados deve ser gerida por um SGBD relacional, sendo acedida via JDBC. Considere, pelo menos, os seguintes tipos básicos de informação, que deverão ser mantidos na BD do sistema. (Note-se que PK denota *Primary Key*)

- Área Científica
  - identificação da área científica (PK)
  - informação geral: nome, objectivos, link para página específica
- Curso
  - identificação do curso (PK)
  - sigla, nome, logótipo
  - objectivos
- Docente
  - identificação do docente (PK)

- nome, morada, link para Web site pessoal, e-mail
  - grau académico (e.g., licenciatura, mestrado, doutoramento)
  - grau na carreira (e.g., ASG, AST, PAX, PAS, PCT)
  - informação para autenticação: login, pwd
  - recebe evento automaticamente (sim/não)
  - mecanismo de recepção dos eventos (e-mail/SMS)
- Disciplina
  - identificação da disciplina (PK)
  - identificação do curso, sigla
  - nome, logótipo
  - identificação da área científica
  - identificação do docente responsável
  - *link* para website da disciplina
- Eventos
  - identificação de evento (PK)
  - informação geral: título, data registo/anúncio, data início, data fim de validade
  - identificação do membro (que regista o evento)
  - descrição do evento/notícia, url para mais detalhes
  - tipo de evento
  - âmbito de disseminação (privado, público)

11.4.4 Funcionalidade do Sistema

Disseminação automática de informação

Qualquer docente pode registar eventos/notícias, sendo relevantes a data em que o evento deverá ser divulgado (data início) bem como a sua data de validade (i.e., a data a partir da qual, a sua divulgação deverá cessar). Adicionalmente, indica o âmbito de disseminação: se ao público geral ou se apenas restrito aos docentes.

A divulgação desses eventos é realizada por duas formas complementares. Por um lado, por consulta dos detalhes dos eventos/notícias

numa determinada zona do WebDEI através de páginas HTML ou WAP. Por outro lado, por disseminação, i.e., por envio de mensagens, sejam elas de e-mail ou SMS (*short message service*), salvaguardando-se necessariamente as questões de acessibilidade. Apenas deverão encontrar-se publicados eventos cujo período de validade não tenha expirado. A disseminação de informação de eventos por mensagem consiste no envio periódico (e.g., mensal, parâmetro a configurar) de uma mensagem que resume os eventos válidos para esse período. Apenas devem ser enviadas mensagens aos docentes que tenham assinalado esse desejo.

Operações Providenciadas pelo Sistema

O WebDEI deverá providenciar várias operações típicas de aplicações congéneres. Descreve-se com as letras A, D e G os tipos de acessos correspondentes aos utilizadores referidos anteriormente (respectiva-mente Anónimo, Docente e Gestor). Todos os utilizadores usam uma interface baseada em HTML (estático ou gerado dinamicamente).

Consultas Gerais

Qualquer utilizador (anónimo ou docente) pode consultar a informação descrita acima. A título de exemplo, enumeram-se algumas das consultas típicas a suportar (com a assinatura dos métodos suportados pelo servidor aplicacional).

- Obter a lista de todas as áreas científicas (A,D)

```
Enumeration obterAreasCientificas()
```

- Obter a lista de todas as disciplinas de uma área científica (A,D)

```
Enumeration obterDisciplinasArea (int
idAreaCientifica)
```

- Obter a lista de todos os docentes de uma área científica (A,D)

```
Enumeration obterDocentes(int idAreaCientifica)
```

- Obter a lista de todas as disciplinas de um curso (A,D)

```
Enumeration obterDisciplinasCurso (int idCurso)
```

- Informações gerais sobre um docente (A,D)

```
Object obterDocente(int idDocente)
```

- Informações gerais sobre uma disciplina (A,D)

```
Object obterDisciplina(int idDisciplina)
```

- Todos os eventos disponíveis (A,D)

```
Enumeration obterEventos (int idMembro)
```

- Todos os eventos disponíveis, de acesso restrito a docentes (D)

```
Enumeration obterEventosPrivadosDocentes()
```

Esta informação será enviada automaticamente por correio electrónico para todos os membros do DEI (que tenham obviamente endereço de e-mail).

Gestão de áreas científicas, cursos, docentes, disciplinas e membros-utilizadores

Operações realizadas pelo gestor do WebDEI: considere-se apenas operações de introdução e remoção. A título de exemplo deverão ser suportadas as seguintes operações (G).

```
int adicionaAreaCientifica(AreaCientifica
areaCientifica)
int adicionaCurso(Curso curso)
int adicionaDocente(Docente docente)
int adicionaDisciplina(Disciplina disciplina)
int adicionaMembro(Membro membro)
```

Esta informação deverá estar necessariamente correlacionada. Por exemplo deverão existir outros métodos para criarem associações entre áreas científicas e disciplinas, ou entre áreas científicas e docentes.

Outras operações

Cada docente (D) deverá poder actualizar a sua “ficha” pessoal. Esta funcionalidade corresponderá à geração e tratamento de um formulário HTML correspondente.

Todos os docentes (D) podem registar eventos numa área definida para o efeito.

```
int adicionaEvento(Objecto evento)
```

11.5 Resolução do Caso de Estudo WebDEI

Apresenta-se de seguida a aplicação do processo ICONIX ao caso de estudo WebDEI, conforme apresentado informalmente na secção anterior.

11.5.1 Análise de Requisitos

Modelo do Domínio: Diagrama de Classes de Alto Nível

Esta primeira actividade consiste na identificação de todos os objectos do mundo real e de todas as relações de generalização, associação e agregação existentes entre esses objectos. O resultado desta actividade é o correspondente diagrama de classes de alto nível. Em geral, as melhores formas para o desenvolvimento desta actividade consistem (1) na leitura e interpretação de documentos, caso existam, que contenham o problema e os requisitos de alto nível; e (2) na realização de um conjunto restrito de entrevistas com utilizadores e peritos do domínio do problema.

Identificação das Classes

Considere-se para o efeito, o seguinte extracto de texto relativo à especificação do sistema WebDEI. Uma prática comum consiste em sublinhar os nomes e substantivos, que correspondem a candidatos a classes (ou nalguns casos, a atributos) no modelo do domínio a conceber.

"O DEI tem por missão promover, organizar e realizar actividades de ensino (licenciaturas, mestrados, doutoramentos, pós-graduações, cursos e/ou seminários de curta duração, de carácter profissionalizante), de investigação e de desenvolvimento, bem como actividades de consultoria ligadas ao tecido social e económico.

O DEI é constituído por um conjunto de membros docentes, os quais se encontram organizado em áreas científicas. Cada área científica é

liderada por um <u>professor</u> e é constituída por vários <u>docentes</u> (<u>professores</u> e <u>assistentes</u>). Adicionalmente, cada <u>área científica</u> é responsável explicitamente por um conjunto de <u>disciplinas</u> que podem ser leccionadas em diferentes <u>licenciaturas</u> ou outros <u>tipos de cursos</u>.

O <u>DEI</u> é responsável ainda pela promoção e organização de diversos <u>eventos</u> de <u>carácter científico e profissional</u> bem como por promover <u>mecanismos de disseminação automática de informação</u>."

Atente-se em alguns dos problemas que este texto levanta. Por exemplo:

- Nome do sistema: e.g., "DEI": é o nome do sistema organizacional onde se encontra o sistema computacional WebDEI. Por conseguinte, por definição, nunca se deve representar através de uma classe todo ou parte do sistema que se pretende tratar.
- Nomes não relevantes ou demasiado vagos: e.g., "tecido social e económico" ou "carácter científico e profissional"; não são relevantes para o sistema.
- Sinónimos: e.g., "actividade de ensino" e "curso" são nomes alternativos; solução: escolher uma alternativa e usá-la consistentemente.
- Falta de informação: e.g., qual é a diferença entre um docente e um professor? E entre um docente e um membro? Vamos assumir de ora em diante que um "professor" é um tipo particular de "docente" que é doutorado, e que todo o docente é um utilizador registado (membro) do WebDEI.
- Falta de informação: e.g., "licenciaturas", "mestrados", "doutoramentos", "pós-graduações", e "cursos/seminários rápidos de carácter profissionalizante" correspondem ou não ao mesmo conceito: "curso"? Vamos assumir neste contexto que sim.

Como resultado da análise do texto anterior, e tendo em conta a discussão prévia, identificamos as seguintes classes: `Utilizador`, `Docente`, `GrauAcadémico`, `Curso`, `TipoCurso`, `AreaCientifica`, `Disciplina`, `Evento`, `ActividadeID`, `ActividadeConsultoria`.

Identificação das Relações

Com base nas classes acima enumeradas, o passo seguinte consiste na identificação das relações entre as mesmas, em particular as relações de generalização e de associação. As generalizações retratam relações de pai-filho ou de especialização de tipos entre as classes envolvidas. As relações de associação são captadas a partir de verbos e expressões verbais identificadas no texto, bem como a partir do conhecimento que existe sobre o domínio do problema. Ilustra-se de seguida um extracto de texto com as formas verbais que deverão indiciar relações de associação entre classes.

"O DEI é constituído por um conjunto de membros docentes, os quais pertencem a áreas científicas. Cada área científica é liderada por um professor e é constituída por vários docentes (professores e assistentes). Adicionalmente, cada área científica é responsável explicitamente por um conjunto de disciplinas que podem ser leccionadas em diferentes licenciaturas ou outros tipos de cursos."

Para além das classes anteriormente identificadas, introduziu-se neste diagrama a classe `ParâmetrosConfiguração` de modo a suportar parâmetros vários do sistema, nomeadamente, a periodicidade da disseminação de eventos.

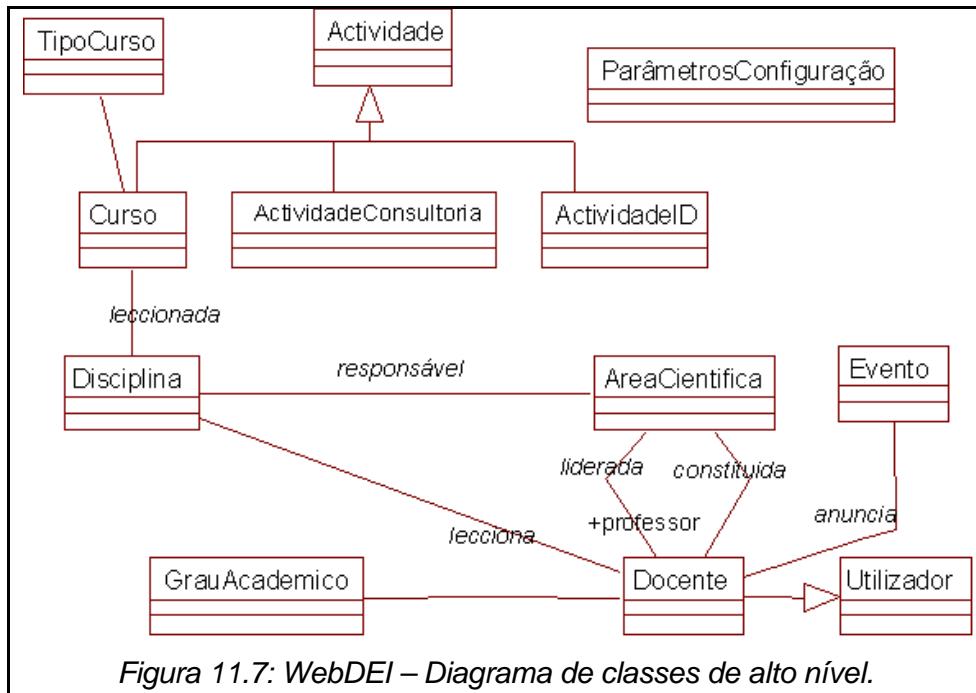
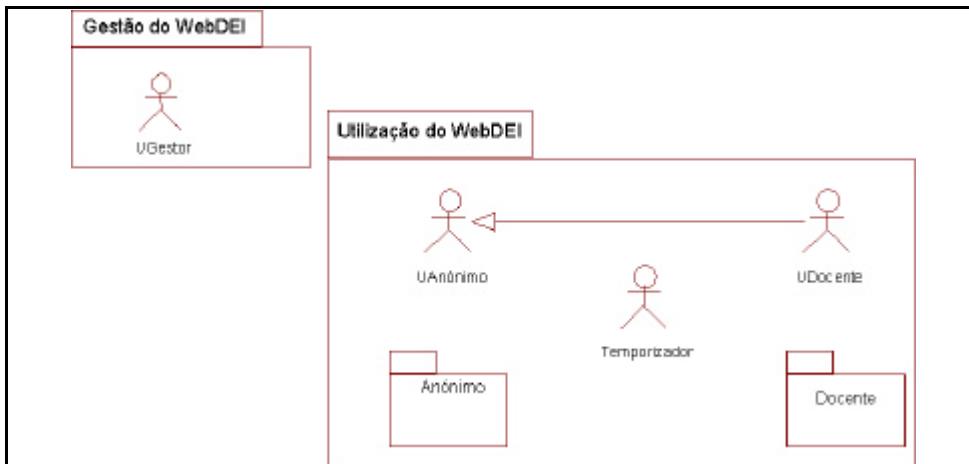



Figura 11.7: WebDEI – Diagrama de classes de alto nível.

Conforme ilustrado na Figura 11.7 não é conveniente, nesta fase, endereçar os aspectos da (1) multiplicidade das relações; (2) especificar se as relações são do tipo agregação ou composição; ou (3) especificar as operações das classes. Por outro lado, a especificação dos atributos das classes já é perfeitamente conveniente (embora não sejam ilustrados na Figura 11.7 por motivos de legibilidade). Tais detalhes são normalmente diferidos para a fase seguinte. A principal vantagem de diferir a especificação desses detalhes para mais tarde, é acelerar a realização da modelação do domínio e evoluir para as fases seguintes.

Prototipagem de GUI

O WebDEI é um sistema com interface Web, pelo que neste contexto e nesta fase pode fazer todo sentido o desenho do seu modelo de navegação, o desenho de algumas páginas HTML representativas, e ainda a especificação de guias de estilos de desenho que deverão ser adoptados consistentemente ao longo de todas as interfaces

produzidas. (Os detalhes desta actividade saem do âmbito deste livro, para mais informações consulte-se por exemplo [Dix98].)

Diagramas de Casos de Utilização

Esta actividade consiste na identificação da funcionalidade do sistema na óptica dos seus utilizadores, o que passa pela identificação dos actores envolvidos, e pela estruturação dos casos de utilização em agrupamentos lógicos, através de diagramas de pacotes.

Figura 11.8: WebDEI – Diagrama de pacotes.

A Figura 11.8 ilustra os diagramas de pacotes assim como os actores identificados para o WebDEI. Definimos dois pacotes principais: o pacote "Gestão do WebDEI", que contém todos os casos que o gestor deverá realizar; e o pacote "Utilização do WebDEI", que contém todos os casos que o utilizador anónimo ("UAnónimo") e o utilizador registado ("Udocente") deverão realizar. Neste último pacote, encontra-se explicitado o actor "Temporizador", necessário para desencadear o caso de utilização associado à disseminação automática de eventos (ver mais à frente). Outro aspecto a destacar é a relação de generalização entre os actores "UAnónimo" e "UDocente", ou seja, o docente é encarado como uma especialização do utilizador anónimo pelo facto de herdar todas as funcionalidades especificadas através do respectivo conjunto de casos de utilização.

As Figuras 11.9, 11.10 e 11.11 apresentam os casos de utilização, respectivamente, do gestor, do utilizador anónimo e do docente do WebDEI.

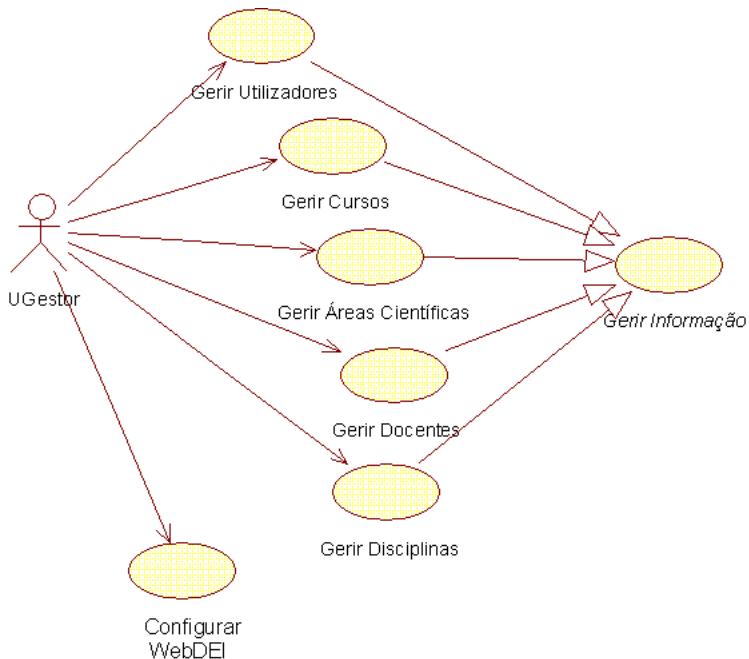
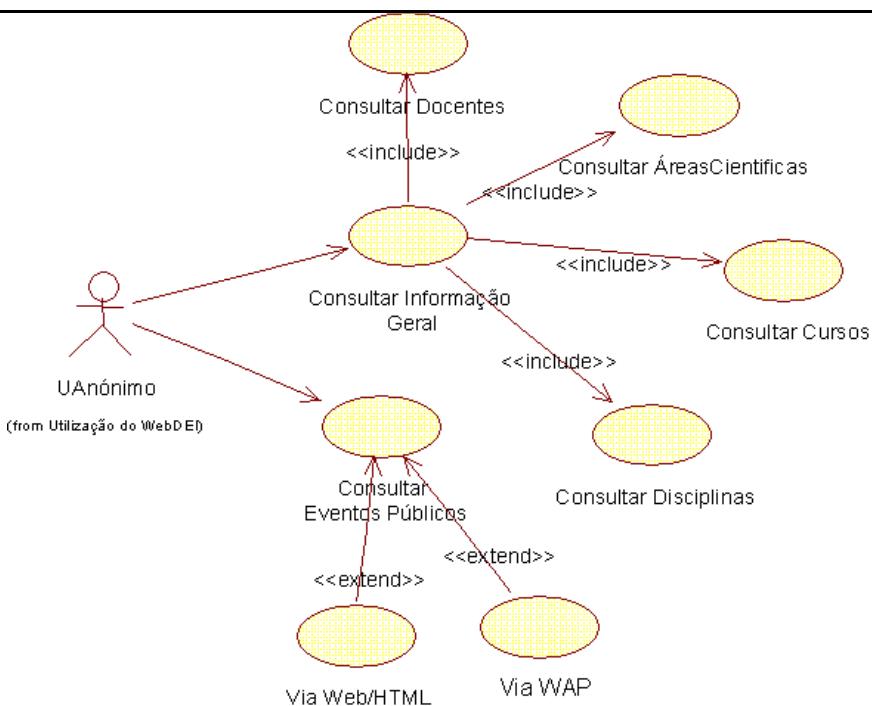



Figura 11.9: WebDEI – Diagrama casos de utilização do gestor.

Figura 11.10: WebDEI – Diagrama casos de utilização do utilizador anónimo.

A Figura 11.10 ilustra os casos de utilização específicos para o utilizador anónimo (e por herança, também para o utilizador-docente). Note-se que, contrariamente ao apresentado na Figura 11.9 (que se adoptou uma abordagem de derivação de casos de utilização a partir de um caso abstracto “Gerir Informação”), neste diagrama existe um caso abrangente “Consultar Informação Geral” que usa a funcionalidade dos diferentes tipos de consultas (e.g., de docentes, de cursos) através de uma relação de inclusão (estereótipo «include»)

Outra observação pertinente deve-se ao facto do ICONIX propor outros tipos de relações entre casos de utilizações. Em vez de usar as relações «include» e «extend» (ver Capítulo 5), o ICONIX propõe a utilização de relações do tipo «precedes» e «invokes». (De qualquer forma, não nos iremos alongar sobre esta discussão e manteremos nos

exemplos seguintes as relações clássicas propostas na especificação oficial do UML.)

O caso “Disseminação de Informação” da Figura 11.11 consiste no envio periódico de informação (ou de um resumo de informação) relativa a todos os eventos que estejam no estado em “divulgação”. Este caso é desencadeado por acção da passagem de tempo, pelo que se representa o actor “Temporizador”. Por outro lado, os “beneficiários” deste caso de utilização são os docentes (que tenham configurado previamente o seu desejo de receberem tais mensagens periódicas). Por fim, note-se a variabilidade deste caso ao permitir dois mecanismos alternativos de envio das mensagens, por e-mail ou por SMS, o qual é representado por dois casos abstractos (i.e., “Via E-mail” e “Via SMS”) que estendem o caso destino no seu ponto de extensão.

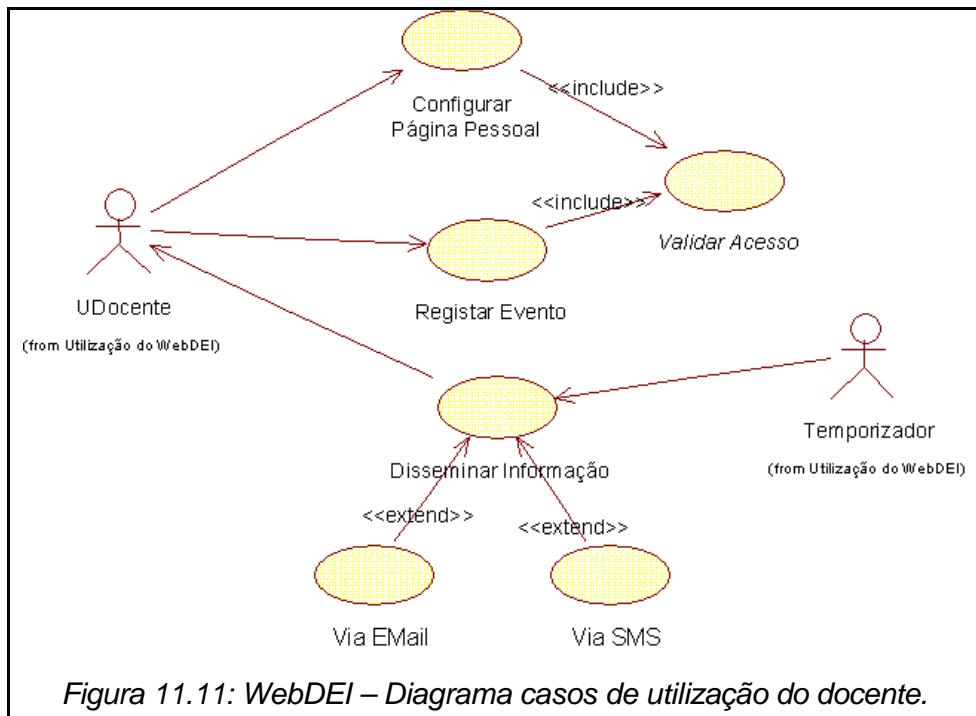



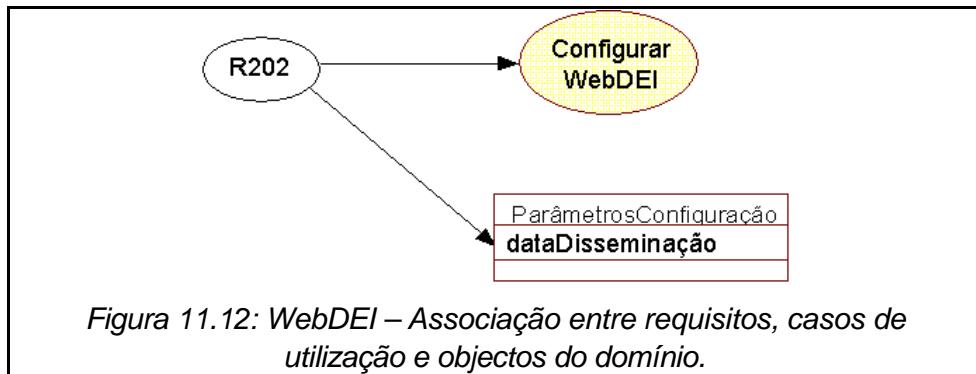
Figura 11.11: WebDEI – Diagrama casos de utilização do docente.

Requisitos Funcionais

No ICONIX, tal como referido anteriormente, requisitos e casos de utilização são conceitos distintos, existindo uma relação de muitos-para-muitos entre si.

A título de exemplo, considere-se a seguinte lista, necessariamente incompleta, de requisitos ("leis que governam o comportamento do sistema") do WebDEI:

R101: O presidente do DEI tem de ser um professor com categoria académica "catedrático" ou "associado".

R102: O coordenador/responsável por qualquer área científica tem de ser professor com categoria académica "auxiliar" ou superior.

R201: Apenas podem receber notificação de eventos os docentes que tenham previamente assinalado o seu desejo, bem como indicado qual o mecanismo de recepção desejado (E-mail ou SMS).

R202: A periodicidade da disseminação dos eventos não pode ser superior a três meses e inferior a uma semana.

R203: Apenas os docentes podem submeter registo de eventos.

…

A Figura 11.12 ilustra a associação entre requisitos, casos de utilização e objectos do domínio, em particular entre o requisito `R202`, o caso de utilização "Configurar WebDEI" e a classe `ParâmetrosConfiguração`. Esta associação revela que o requisito `R202` tem impacto na descrição do caso "Configurar WebDEI" assim como no comportamento da classe `ParâmetrosConfiguração`.

Figura 11.12: WebDEI – Associação entre requisitos, casos de utilização e objectos do domínio.

11.5.2 Análise e Desenho Preliminar

Detalhe dos Casos de Utilização

A primeira actividade da tarefa de análise e desenho preliminar consiste em detalhar os casos de utilização anteriormente identificados. Para tal, são descritos textualmente todos os cenários correspondentes a cada caso de utilização, sendo em geral contemplados os cenários principais, alternativos e de excepção.

Nesta actividade, a principal sugestão do ICONIX é que não se deve despender muito tempo com a descrição textual, e que se pode usar um estilo de escrita qualquer, que seja conveniente no contexto do projecto, desde que usado consistentemente.

Como exemplo, vamos apresentar a descrição textual de apenas um caso de utilização, designadamente do caso "Registar Evento".

Nome: ***Registar Evento***

Actores: Udocente

Objectivo: Registar o anúncio de um evento (e.g., conferência, reunião, notícia) relevante para a comunidade do DEI.

Cenário Principal (descrição de alto nível):

1) Incluir o caso de utilização "Validar Acesso".

2) O utilizador-docente (UDocente), após a sua validação pelo sistema, recebe do sistema o formulário "Registo de Evento".

3) O UDocente preenche adequadamente o formulário recebido, em particular especificando: o título; uma breve descrição; um url para mais detalhes; a data de início e de fim do evento; e específica o âmbito da sua divulgação (se âmbito público, se âmbito restrito (apenas para docentes)).

4) O UDocente submete a informação introduzida.

5) O sistema recebe a informação introduzida, valida-a (por exemplo, não permitindo a introdução de dois eventos com o mesmo título), e guarda a informação numa base de dados. Para além da informação introduzida pelo UDocente, o sistema adiciona ainda a data de registo e a identificação do docente interveniente.

6) O sistema notifica adequadamente o UDocente que o registo foi realizado com sucesso.

Cenário Alternativo 1 (Desistência de preenchimento do formulário):

Idem aos passos 1), 2) e 3) do Fluxo Principal.

4) O UDocente não submete o formulário num período de 1 hora, a partir do momento que o formulário foi enviado pelo sistema. O sistema aborta a transação e o caso de utilização é reinicializado.

Cenário Alternativo 2 (Informação inválida):

Idem aos passos 1), 2) 3) e 4) do Fluxo Principal.

5) O sistema recebe a informação introduzido, mas esta não passa nos testes de validação.

6) O sistema notifica adequadamente o UDocente que o registo não foi realizado.

Análise de Robustez

A análise de robustez é uma actividade importante no ICONIX. Este tipo de actividade foi inicialmente proposto por Ivar Jacobson, de forma a permitir ilustrar graficamente as interacções entre objectos participantes num caso de utilização. Foram definidos três tipos de objectos, os quais se encontram definidos no perfil "Processos de Desenvolvimento de Software" especificado no UML 1.3 (ver Secção 9.4.1):

- Objectos de fronteira/interface («boundary»), permitem aos actores comunicarem com o sistema. Exemplos comuns deste tipo de objecto são janelas, ecrãs, páginas Web, janelas de diálogo.
- Objectos de entidade («entity»), correspondem geralmente aos objectos identificados no modelo do domínio. Estes objectos são geralmente mapeados em tabelas de bases de dados ou ficheiros que guardam a informação necessária.
- Objectos de controlo («control»), funcionam como integradores entre os objectos de fronteira e os objectos de entidade. O objectivo destes objectos é conterem as regras de negócio e as políticas de funcionamento de modo a potenciarem a independência das interfaces com os utilizadores, por um lado, e dos esquemas das bases de dados, por outro. Estes objectos terminam ocasionalmente como objectos no modelo estático; mas mais geralmente, acabam por ser convertidos em métodos de objectos de entidade ou de objectos de fronteira.

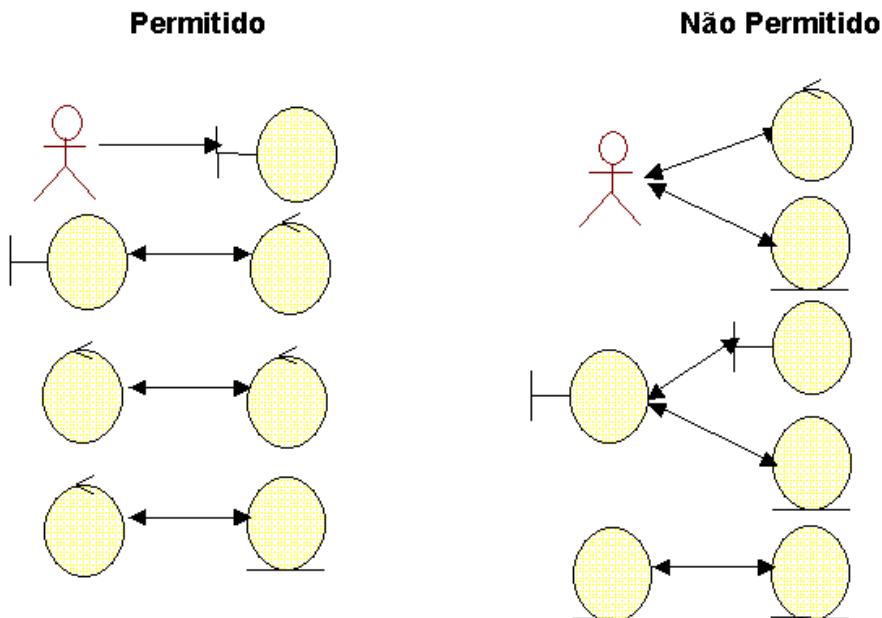



Figura 11.13: Tipos de objectos usados os diagramas de robustez.

A análise de robustez desempenha no ICONIX um papel charneira na passagem entre a análise ("o quê") e o desenho ("o como"), tendo as seguintes funções principais:

- Teste de razoabilidade: permite validar se a descrição textual de um caso de utilização tem sentido e se a especificação do comportamento do sistema é razoável, dado o conjunto de objectos identificados.
- Teste de completude: permite validar se todos os cursos de execução de um sistema são descritos através de casos de utilização.
- Descoberta de objectos: permite identificar novos objectos que não foram descobertos anteriormente durante a modelação do domínio.

Uma sugestão do ICONIX é desenvolver os diagramas de análise de robustez antes, ou em paralelo, com a descrição textual dos casos de utilização, de modo a influenciar a identificação dos objectos intervenientes e a escolha dos nomes usados. A ideia geral é que os nomes genéricos deverão ser substituídos por nomes de objectos que aparecem nos diagramas de robustez. Outra sugestão relevante é evitar fazer desenho detalhado nesta fase e nestes tipos de diagramas. O seu principal objectivo é captar, para cada caso de utilização, os principais objectos e respectivas relações de comunicação estabelecidas entre os mesmos.

Figura 11.14: Regras de desenho dos diagramas de robustez.

O ICONIX preconiza um conjunto de regras para auxiliar o desenho de diagramas de robustez, em particular são definidas as seguintes regras (ver Figura 11.14):

- Os actores podem comunicar com o sistema através de objectos fronteira.
- Os objectos fronteira comunicam apenas com actores e objectos de controlo.
- Os objectos entidade comunicam apenas com objectos de controlo.
- Os objectos de controlo comunicam apenas com objectos de fronteira e de entidade.

Apresenta-se na Figura 11.15, a título de exemplo, o diagrama de robustez relativo ao caso de utilização "Registar Evento". Note-se nas interacções entre os diferentes tipos de objectos, que traduzem em termos gerais o comportamento deste caso. Observe-se ainda as relações de agregação entre a página de registo de eventos e os seus respectivos botões.

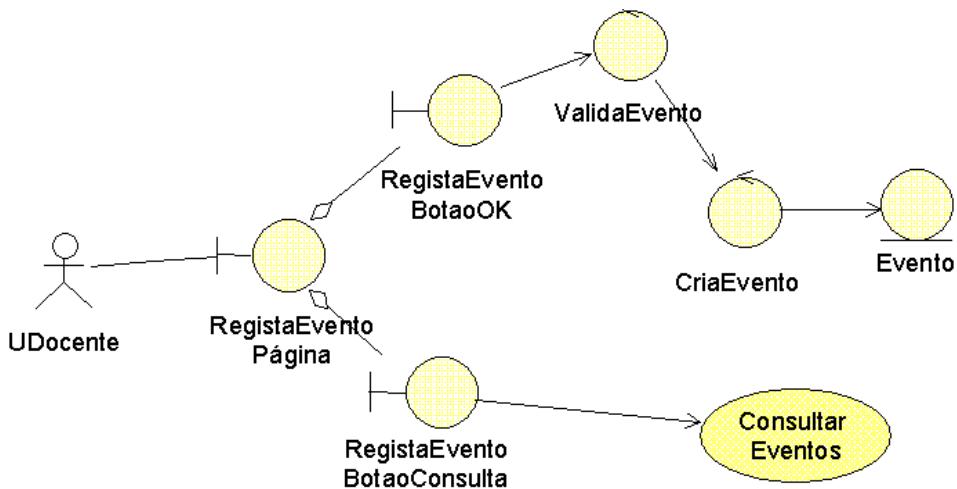
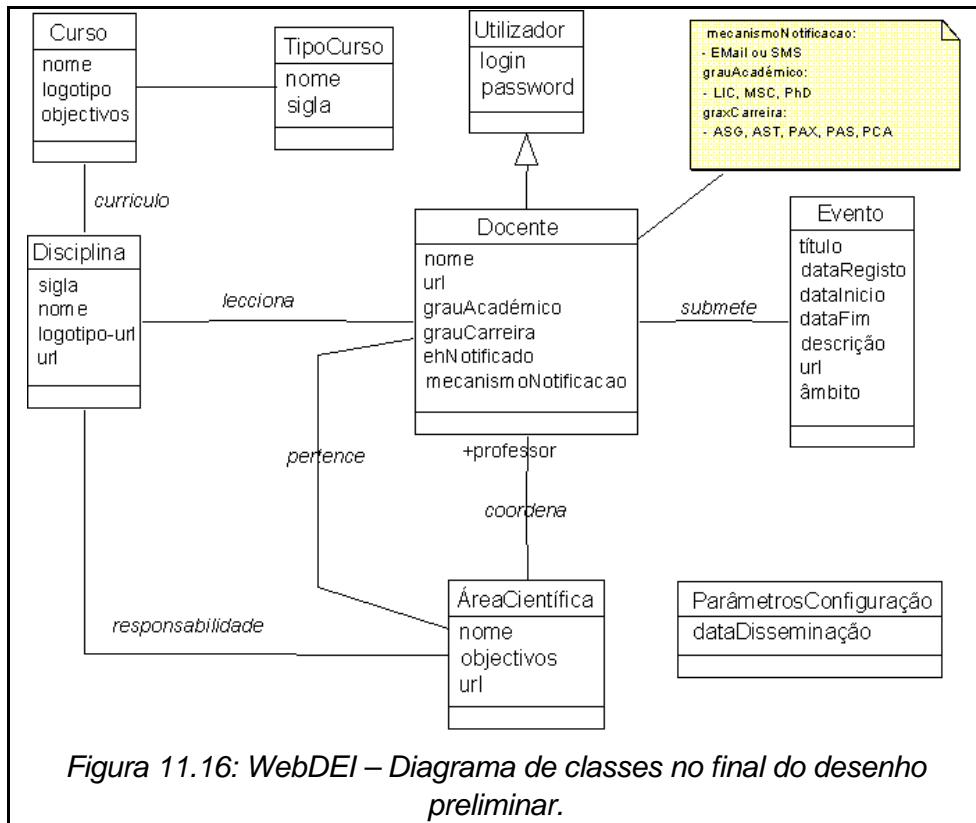



Figura 11.15: WebDEI – Diagrama de robustez do caso "Registar Evento".

Outro aspecto ilustrado na Figura 11.15, que o ICONIX sugere, é a possibilidade de representação explicita da invocação de um caso de utilização (e.g., "Consultar Eventos") a partir de um objecto fronteira.

Actualizar o Diagrama de Classes

Na realização dos diagramas de robustez, por cada caso de utilização, foram usadas classes anteriormente definidas (e.g., a classe `Evento`), mas foram também descobertas novas classes (e.g., `RegistaEventoPágina`, `ValidaEvento`), assim como atributos, que deverão actualizar o diagrama de classes anteriormente construído.

O mecanismo de actualização do diagrama de classes deve ser realizado de forma cuidadosa de modo a evitar introduzir-se classes que mais tarde serão transformadas em métodos (e.g., a classe `ValidaEvento`).

Nesta fase, percebemos também que não há qualquer caso de utilização relacionado com as classes `ActividadeConsultoria` e `ActividadeID`, pelo que as iremos eliminar do diagrama de classes original. A Figura 11.16 ilustra o diagrama de classes resultante desta actividade. Por motivos de simplicidade e facilidade de leitura não se introduziram as outras classes correspondentes aos objectos de fronteira identificados nos diagramas de robustez.

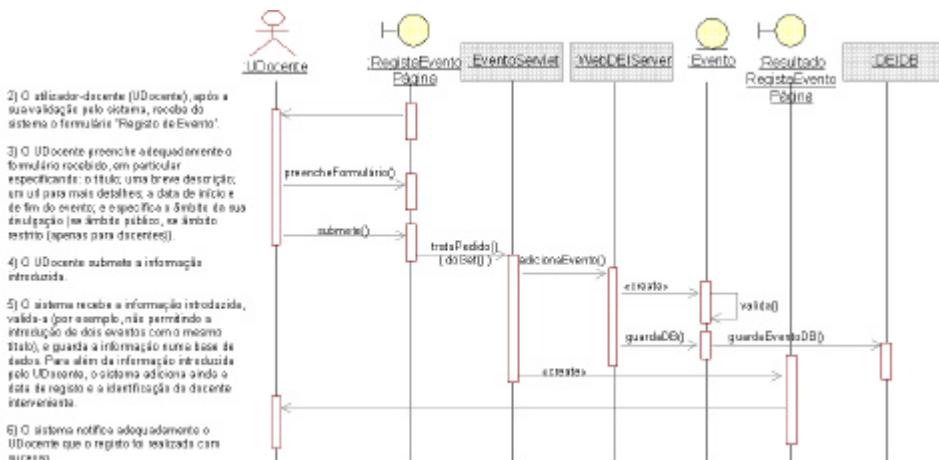
Figura 11.16: WebDEI – Diagrama de classes no final do desenho preliminar.

11.5.3 Desenho

O principal objectivo desta actividade é detalhar o desenho do sistema tendo em consideração a infra-estrutura computacional de suporte e a tecnologia de desenvolvimento envolvida.

Neste caso de estudo, a infra-estrutura computacional corresponde à arquitectura a quatro camadas descrita no enunciado (ver Secção 11.4), e a tecnologia de desenvolvimento consiste num conjunto de páginas HTML geradas dinamicamente a partir de servlets Java, numa aplicação servidor multiactividade desenvolvida em Java, e numa base de dados suportada por um gestor de base de dados não determinado à priori. Complementarmente, o cliente webcomunica com o servidor Web e os *servlets* através do protocolo HTTP, os *servlets* comunicam com o servidor Java através do protocolo RMI (*Java Remote Method Invocation*), e o servidor comunica com o gestor de base de dados através do JDBC (*Java Data Base Connectivity*).

Especificar o Comportamento

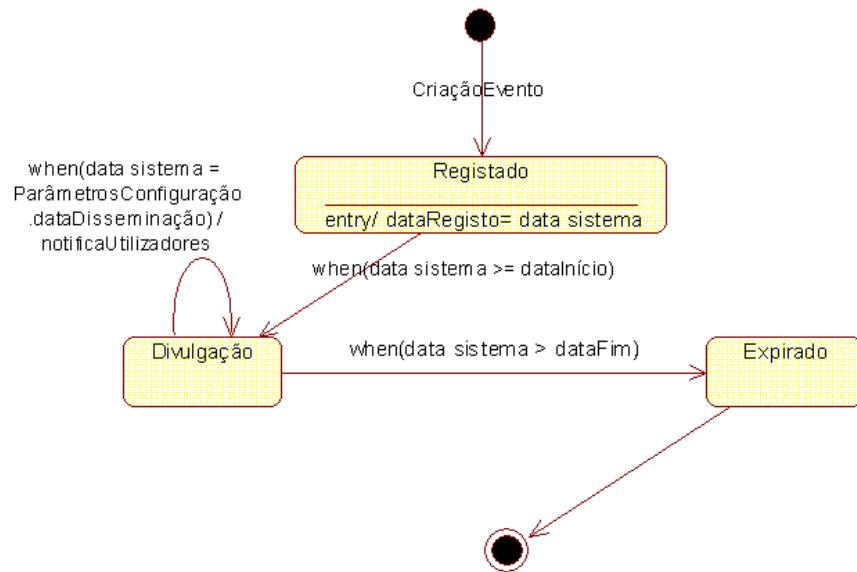
A primeira actividade desta tarefa consiste na especificação do comportamento do sistema. Tal especificação é conduzida pelos casos de utilização anteriormente identificados e descritos através dos respectivos diagramas de robustez e descrições textuais.

O comportamento de um caso de utilização especificado anteriormente através de um diagrama de robustez é agora detalhado através de um diagrama de sequência. Este diagrama deve usar a generalidade dos objectos e actores representados no diagrama de robustez, mas agora evidenciando o fluxo de mensagens trocadas entre si.

O ICONIX sugere a seguinte sequências de passos:

1. Copiar o texto do caso de utilização para a margem esquerda do diagrama de sequência.

2. Adicionar os objectos de entidade.
3. Adicionar os objectos de fronteira.
4. Analisar e descobrir para cada objecto de controlo, em que objectos o seu comportamento deve ser atribuído. (Ou seja, em geral, não se representam os objectos de controlo, já que o seu comportamento corresponde a operações suportadas pelos outros objectos participantes.)

No caso do WebDEI, vamos assumir que a validação e a criação de um evento será realizada no contexto do servidor Java (e.g., classe `DEIServer`), na classe `Evento`. Assumimos ainda que as páginas HTML são geradas dinamicamente a partir de servlets correspondentes (e.g., classe `EventoServlet`). A Figura 11.17 ilustra o diagrama de sequência relativo ao caso de utilização "Registar Evento".

Figura 11.17: WebDEI – Diagrama de sequência do caso "Registar Evento".

Note-se que o diagrama ilustra também a relação entre a classe `Evento` e a hipotética classe `DEIDB` que sabe guardar os diferentes objectos geridos no servidor Java. Esta última classe esconde

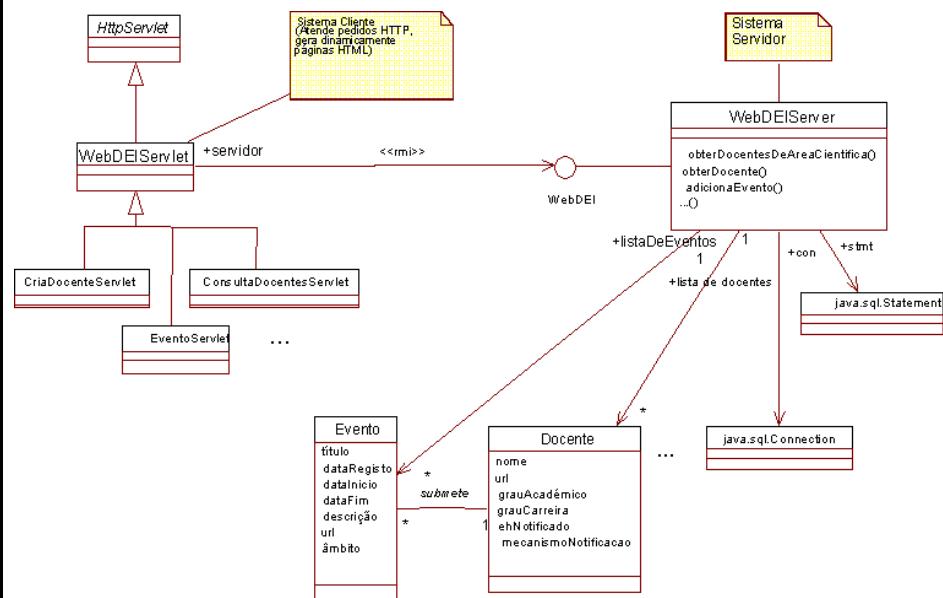
propositadamente todos os detalhes das relações entre o servidor Java e o gestor de base de dados.

Isto é um exemplo hipotético, mas ilustra claramente algumas decisões de desenho, que terão impacto importante ao nível da implementação. Entre outros, um aspecto relevante tem a ver com o facto das classes responsáveis pela produção da interface com o utilizador, os *servlets*, não usarem directamente as classes de entidade (como por exemplo, a classe `Evento`) o que implica que a comunicação entre os *servlets* e o servidor Java será sobre RMI mas com um protocolo de baixo nível, por exemplo por troca de *strings*, em vez de troca de objectos serializados.

Nesta actividade de especificação do comportamento, e se for relevante, pode-se usar também diagramas de colaboração para ilustrar as transacções principais entre objectos, em particular em situações com padrões de desenho. Pode também ser considerado nesta fase o desenho de diagramas de estados ou de diagramas de actividade. Contudo, este tipo de diagramas não são particularmente importantes no ICONIX. Em particular o ICONIX dá ênfase à utilização de diagramas de estados associados a casos de utilização em vez de, como é mais comum, a objectos/classes.

A Figura 12.18 ilustra, a título de exemplo, o diagrama de estados correspondente à classe `Evento`.

Figura 11.18: WebDEI – diagrama de estados da classe "Evento".

Um evento passa ao longo da sua vida por três estados. Após a sua criação fica no estado "Registado". Quando a data de início de divulgação do evento é igual ou superior à data corrente (i.e., à data do sistema), o evento evoluie para o estado "Em Divulgação", podendo ser então consultado via Web. Ainda nesse estado, a informação sobre o evento é enviada periodicamente segundo o parâmetro `dataDisseminação` definido na classe `ParâmetrosConfiguração`. Por fim, quando se atinge a data de fim do evento, este passa para o estado "Expirado".

Terminar o Modelo Estático

A segunda actividade desta tarefa consiste em terminar o modelo estático, adicionando informação detalhada com base nos diagramas de interacção que entretanto tenham sido desenvolvidos, que obviamente identificaram as operações que deverão fazer parte das classes

correspondentes. Adicionalmente, o diagrama de classes deverá incluir todos os atributos, operações, adornos de visibilidade e de navegabilidade.

A definição e utilização de padrões de desenho de forma consistente é também corrente nesta fase final do desenho dos diagramas de classe.

A Figura 11.19 apresenta um esboço relevante do diagrama de classes do WebDEI, resultante desta actividade de desenho.

Figura 11.19: WebDEI – esboço do diagrama de classes da fase de desenho.

Alguns aspectos merecem ser referidos relativamente ao diagrama da Figura 11.19:

- O servidor Java consiste na classe `WebDEIServer`, que mantêm estruturas de dados (no diagrama não está explícito que tipos de estruturas de dados são definidas, se listas, se tabelas de acesso associativo (*hashtables*), se vectores dinâmicos, etc.) com os principais objectos identificados no modelo do domínio.
- O servidor Java providencia um ponto de acesso, via RMI, ao implementar e publicar a interface remota `WebDEI`.

- O servidor Java é responsável por estabelecer conexões, via JDBC, com a base de dados `DEIDB`; para tal mantém variáveis internas do tipo `java.sql.Connection` e `java.sql.Statement`.
- Os *servlets* são derivados a partir da classe abstracta `WebDEIServlet` que impõe um determinado estilo arquitectural, em particular contêm uma referência para o servidor Java (váriável `servidor`), através da interface remota `WebDEI`.

Verificar Requisitos

Para terminar a tarefa de desenho, ou por outra forma, antes de se iniciar a implementação propriamente dita, o ICONIX sugere que se verifique se o desenho satisfaz todos os requisitos identificados. Em particular, o ICONIX sugere a aplicação da seguinte metodologia:

1. Produzir a lista de requisitos.
2. Escrever o manual de utilizador do sistema, na forma de casos de utilização.
3. Iterar com os utilizadores e clientes até se conseguir "fechar" os itens 1 e 2.
4. Certificar que se consegue determinar, para cada requisito, o seu impacto em que parte do desenho, e vice-versa.
5. Determinar, a partir das diferentes partes do desenho, que requisitos estão envolvidos.

11.5.4 Implementação

O ICONIX refere explicitamente que considera a tarefa de implementação como fora do seu âmbito e foco de interesse. Não deixa contudo de providenciar um número restrito de sugestões que merecem alguma referência.

Designadamente, o ICONIX tece um conjunto de observações relativas à atribuição de tarefas e actividades a diferentes intervenientes do processo. Por exemplo, sugere (1) que os casos de utilização sejam

desenvolvidos por indivíduos com experiência no desenho de GUI ou por escritores técnicos com experiência na produção de manuais de utilizador; (2) que os modelos de domínio e diagramas detalhados de classes sejam desenvolvidos por indivíduos com experiência no desenho de bases de dados; (3) que os programadores de sistema devem focar-se em aspectos tais como desempenho, segurança, e serem os responsáveis pelo desenho dos diagramas de estados e de colaboração. É sugerido que a tarefa do desenho detalhado (em particular o desenho de diagramas de sequência) seja realizado, ou pelo menos supervisionado, por indivíduos seniores com experiência em modelação OO.

O ICONIX refere a possibilidade de se desenharem nesta fase diagramas de arquitectura de forma a clarificarem alguns detalhes da instalação e da implementação propriamente dita. Por exemplo, o ICONIX sugere a utilização de diagrama de componentes para se ilustrar o mapeamento entre as classes e os componentes, ou seja, para ilustrar como é que as classes se deverão encontrar organizadas através da noção de componentes de trabalho. Por exemplo, pode-se explicitar, para cada componente, que classes é que aquele implementa.

A Figura 11.20 ilustra o diagrama de componentes de instalação constituinte do WebDEI. Note-se em particular a explicitação da sua arquitectura a quatro camadas: cliente universal, UI (*user interface*), lógica e dados; tal como as relações existentes entre os diversos componentes do sistema: http, rmi e jdbc. (Sugere-se ao leitor mais atento, uma análise e comparação entre as Figuras 12.20 e 12.6 esperando que as conclusões sejam, neste momento, óbvias!)

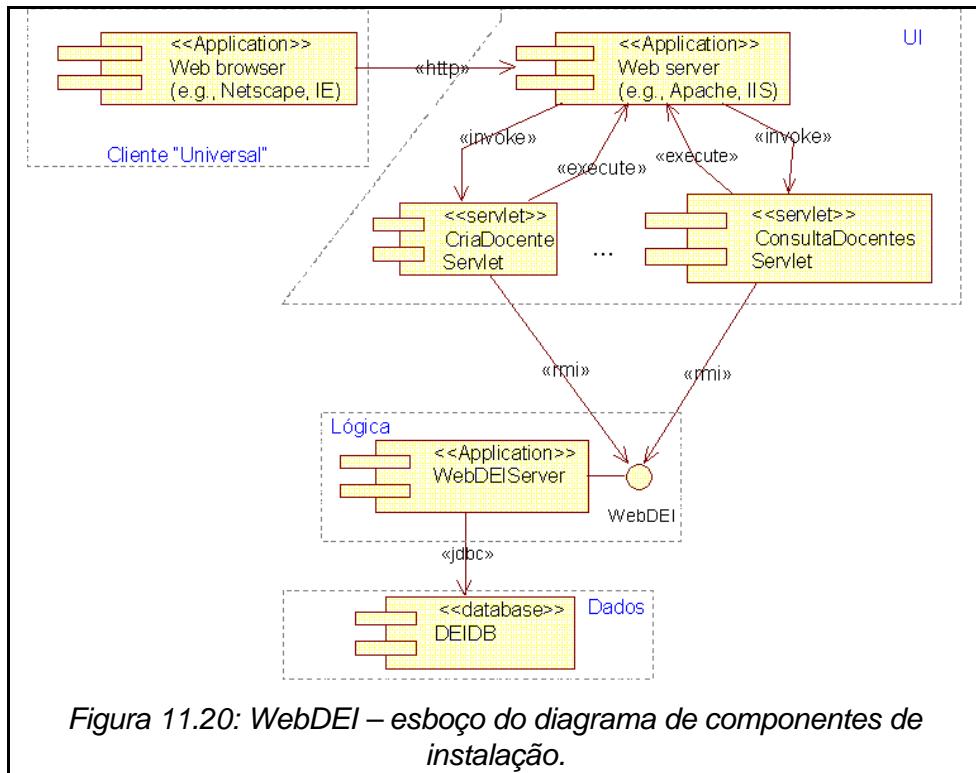



Figura 11.20: WebDEI – esboço do diagrama de componentes de instalação.

Sai fora do âmbito do livro a discussão e análise dos detalhes de implementação e codificação deste caso de estudo. Também não foi nosso propósito, neste livro, realizar o documento de análise e desenho do sistema WebDEI na sua totalidade; mas apenas usar o exemplo WebDEI para ilustrar e discutir a aplicação do processo ICONIX.

11.6 Conclusão

Neste capítulo respondemos à questão colocada anteriormente: como usar o UML no processo de desenvolvimento de software? Apresentámos o ICONIX, que é como vimos um processo iterativo e incremental, conduzido por casos de utilização e com modelação visual baseada em UML.

Podemos dizer que o ICONIX é um processo que, através de uma abordagem essencialmente prática, ensina a modelar um sistema de

software segundo o paradigma OO. O objectivo global do ICONIX é, a partir de um conjunto de requisitos inicialmente definido, a construção do modelo de classes que suporte a implementação de determinado sistema.

Neste processo, sem dúvida que os seus aspectos principais consistem na identificação e representação do modelo do domínio (uma versão inicial e de alto nível do diagrama de classes) e dos casos de utilização. A partir destes dois modelos tudo se desenrola iterativa e incrementalmente: cada caso de utilização é detalhado através de uma descrição textual e respectivo diagrama de robustez. São identificados novos objectos e detalhes, os quais são adicionados ao diagrama de classes. Seguidamente desenham-se os diagramas de sequência (com um significativo nível de detalhe) identificando-se o comportamento (i.e., as operações) dos objectos intervenientes. Essas operações são adicionadas numa versão detalhada e final dos diagramas de classes.

A ideia chave do ICONIX poder-se-á identificar como “fazer (i.e. modelar) o menos possível, no mais curto período de tempo, de forma a concretizar um bom sistema”. O ICONIX não privilegia explicitamente a utilização de vários diagramas UML, em particular não privilegia os diagramas de estado, de actividade, de arquitectura, e mesmo os diagramas de colaboração. Tal posição parece-nos adequada num processo que se pretende prático e simples. No entanto, outros processos igualmente simples e práticos sugerem a utilização de outros diagramas do UML. Por exemplo, o GRAPPLE [McConnell96] sugere explicitamente o desenho de diagramas de actividades para modelar processos de negócio na fase preliminar do processo de software, o que na nossa opinião e experiência pode ser benéfico.

Outra ideia forte do ICONIX é a distinção entre requisitos e casos de utilização. Para os seus autores, um requisito é um requisito e um caso de utilização é um caso de utilização. São conceitos distintos, existindo uma relação de muitos-para-muitos entre si. Esta posição vai no sentido contrário de várias outras propostas, que consideram que os requisitos de um sistema podem ser representados e mantidos, com significativos benefícios, através dos casos de utilização. Uma desvantagem óbvia desta posição do ICONIX é obrigar à equipa do projecto a identificação e gestão de listas de requisitos, assim como à gestão das relações

entre requisitos e casos de utilização, o que implica um esforço e volume de trabalho mais significativo, o que na nossa opinião deveria ser evitado num processo que se pretende rápido e simples.

11.7 Exercícios

Ex.78. Tendo em conta os seus conhecimentos sobre o ICONIX, quais são os tipos de empresas que maior partido poderão tirar desta metodologia de desenvolvimento?

Ex.79. Quais são as principais características do ICONIX?

Ex.80. Quais são os diagramas UML cuja aplicação não é relevante segundo o ICONIX?

Ex.81. Quais são os principais tipos de objectos usados na actividade de análise de robustez?

Ex.82. O que são "diagramas de robustez"? Qual a sua importância no contexto do ICONIX?

Ex.83. Qual é a tomada de posição do ICONIX relativamente à relação entre casos de utilização e requisitos? Discuta justificadamente se concorda ou não com essa posição.

Ex.84. Faça a descrição textual dos casos de utilização apresentados na Figura 11.11, relativamente ao caso de estudo WebDEI.

Ex.85. Qual é a sequência de passos preconizada no ICONIX para se passar de um diagrama de robustez (conjuntamente com a descrição textual do caso de utilização) para um diagrama de sequência?

Ex.86. Modifique o diagrama de classes da fase de desenho, apresentado na Figura 11.19, relativamente ao caso de estudo WebDEI, considerando que tanto os *servlets* (derivados a partir da classe `WebDEIServlet`) como o `WebDEIServer` têm acesso às classes de entidade (e.g., `Evento`, `Docente`, `AreaCientifica`), e que em vez de trocarem strings, trocas instâncias dessas classes serializadas.

Parte 4

Ferramentas CASE

A maioria das pessoas presume que o mundo que conhece durará indefinidamente. Essas pessoas têm dificuldade em imaginar um modo de vida verdadeiramente diferente para elas próprias, quanto mais uma civilização totalmente nova. Claro que reconhecem que as coisas estão a mudar. Mas presumem que as mudanças presentes passarão de qualquer modo sem lhes tocar e que nada abalará o quadro económico familiar nem a estrutura política. Esperam confiantemente que o futuro continue o presente.

Alvin Toffler, A Terceira Vaga, 1980.

As Ferramentas CASE

A tradicional expressão "em casa de ferreiro espeto de pau" pode aplicar-se com alguma propriedade aos informáticos: tal como o ferreiro não utilizava utensílios para facilitar o seu trabalho, uma parte significativa da actividade de desenvolvimento de software foi sempre

realizada com apoio reduzido de ferramentas. Nos últimos anos tem-se procurado corrigir esta situação, e um bom exemplo disso é a utilização de ambientes de desenvolvimento integrados. No entanto, a maioria destes ambientes concentram a sua atenção na fase de implementação do software, proporcionando um reduzido suporte à sua fase de concepção.

Quando surgiu a ideia de escrever este livro, a estrutura que nos pareceu mais adequada apontava para a organização do livro em quatro partes: (1) uma introdução, onde fossem analisadas as questões mais abrangentes da engenharia de software, de modo a enquadrar o leitor; (2) apresentação detalhada da linguagem UML; (3) descrição do impacto do UML ao nível das metodologias de desenvolvimento de software e (4) utilização do UML em ferramentas de modelação.

O objectivo da Parte 4 é esclarecer um conjunto de conceitos genéricos e o âmbito de intervenção destas ferramentas de modelação. De forma a concretizar alguns destes conceitos, e tendo em conta que o livro incide sobre o UML, procurámos identificar no mercado algumas ferramentas que utilizam de base esta linguagem de modelação.

Os produtos seleccionados foram o Rose, por ser a ferramenta de modelação da empresa Rational, onde foi produzida a especificação do UML, e o System Architect, uma ferramenta existente no mercado há muito tempo. e numa posição destacada, que suporta outras técnicas de modelação, mas que actualmente disponibiliza também modelação em UML. Para além destes critérios, outros foram utilizados:

- O pressuposto de que gostaríamos de apresentar uma ferramenta UML pura, isto é, que apenas (ou quase) suporta o UML como técnica de modelação (o caso do Rose) e outra que tivesse um passado de sucesso na aplicação das notações e metodologias estruturadas (o caso do System Architect).
- A preocupação de seleccionar ferramentas líderes do mercado de modelação, como são reconhecidamente o Rose e o System Architect.
- A facilidade da obtenção das respectivas licenças para a maioria dos utilizadores, isto é, a preocupação de seleccionar ferramentas que possam ser utilizadas em computadores pessoais.

Os capítulos relativos às duas ferramentas analisadas não pretendem substituir os manuais dos produtos, pelo que o leitor não irá aqui encontrar informação detalhada sobre o conteúdo dos menus ou da *toolbar* respectiva, ou uma descrição de todas as opções disponíveis. Igualmente, também não pretendem ser uma avaliação completa e formal de cada produto, pois não vamos analisar muitos aspectos que para tal seriam relevantes (por exemplo, os passos do processo de instalação da ferramenta, mecanismos de suporte disponibilizados pelo fornecedor, desempenho do produto).

O nosso objectivo principal com estes dois últimos capítulos do livro é apresentar duas ferramentas que têm concentrado a sua atenção em técnicas de modelação, e em particular, no UML. Nesses capítulos demonstra-se como é que as duas ferramentas seleccionadas se comportam em relação a alguns critérios, que poderão ser factores diferenciadores numa decisão de utilização e/ou aquisição, e dos quais destacamos os seguintes:

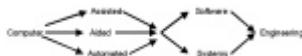
- Modelação de processos de negócio.
- Geração automática de documentação.
- Processo de *round-trip engineering*, quer para o código de aplicações quer para a geração de *scripts* de bases de dados.
- Mecanismos de extensibilidade disponibilizados pela ferramenta.

Organização da Parte 4

O Capítulo 12, “Ferramentas CASE”, tem por objectivo definir o(s) significado(s) do acrónimo CASE, apresentar diversos conceitos teóricos, descrever a evolução deste tipo de ferramentas e introduzir uma taxonomia das ferramentas CASE. São também referidas as funcionalidades mais relevantes que as ferramentas de modelação providenciam e abordadas algumas questões importantes que se colocam às ferramentas de modelação em UML.

O Capítulo 13, “Rose”, descreve resumidamente a evolução e as características gerais do *Rose*, e de seguida apresenta mais detalhadamente, recorrendo a exemplos práticos, a forma como este produto resolve algumas questões importantes neste tipo de ferramentas, com UML.

O Capítulo 14, "System Architect", segue uma estrutura idêntica ao capítulo anterior, se bem que pelo facto de utilizar outras técnicas de modelação para além do UML, estas também serão brevemente analisadas, até porque nalguns casos o System Architect utiliza-as para resolver alguns problemas sem recorrer ao UML.

Capítulo 12 - FERRAMENTAS CASE

Tópicos

- Introdução
- Evolução Histórica
- Arquitectura das Ferramentas CASE
- Mecanismos de Integração entre Ferramentas
- Taxonomia das Ferramentas CASE
- Vantagens e Problemas das Ferramentas CASE
- Funcionalidades das Ferramentas CASE
- Geração Automática de Artefactos
- Avaliação de Ferramentas CASE
- Ferramentas de Modelação para UML
- Conclusão
- Exercícios

12.1 Introdução

O acrónimo CASE tem tido, ao longo do tempo, diversas interpretações, se bem que a mais utilizada e referida na literatura é a de *Computer Aided Software Engineering*, o que, numa tradução literal, significa "Engenharia de Software Auxiliada por Computador". Se relativamente ao significado das letras iniciais "C" e "E" existe algum consenso, o mesmo não acontece com as outras duas letras, cujos diferentes significados podem ser observados Figura 12.1.

Figura 12.1: Significado do Acrónimo CASE.

A letra “A” tem sido utilizada frequentemente para significar também *Assisted* ou *Automated*; no primeiro caso, a palavra pode ser encarada como sinónimo de *Aided,* enquanto que no segundo caso a ideia é realçar a importância da utilização de meios informáticos nas actividades relacionadas com a engenharia de software, de modo a que essas actividades sejam realizadas de forma automática. No entanto, e independentemente da utilização de qualquer dos três significados, a distinção não é neste caso muito significativa.

A maior divergência tem a ver com o significado da letra “S”, uma vez que a alternativa é *Systems*. A palavra "Sistema" dá uma abrangência maior ao conceito CASE, uma vez que passaríamos a incluir não apenas questões relacionadas com a Engenharia de Software, e portanto cujo objectivo último tem a ver com o desenvolvimento de software, mas também aspectos relacionados com as questões do Sistema de Informação no seu todo, e que no Capítulo 1 incluímos no âmbito do Planeamento Estratégico de Sistemas de Informação e da definição da Arquitectura de Sistemas de Informação.

Na nossa perspectiva, mais importante do que o significado das quatro letras é a definição do que se entende por CASE. Numa primeira análise, o termo CASE não fala explicitamente de ferramentas, mas se pensarmos na expressão "auxiliado por computador", tal só pode ser conseguido pela utilização de ferramentas informáticas.

Antes de apresentamos a nossa definição do conceito CASE, vale a pena constatarmos as opiniões que outros "pensadores" sobre o assunto emitiram anteriormente:

- B. Terry [Terry90]: "CASE designa um conjunto de ferramentas que auxiliam um programador ou um gestor de projectos durante uma ou mais fases do processo de desenvolvimento de software, incluindo a manutenção".

- Carma McClure [McClure89]: "Uma combinação de ferramentas de software e de metodologias de desenvolvimento estruturadas". Esta definição reflecte claramente o momento histórico em que foi elaborada, uma vez que utiliza a expressão "metodologias estruturadas" para definir CASE, pois à data em que a definição foi elaborada, não existiam outras metodologias com expressão.
- Software Engineering Institute (*www.sei.cmu.edu*): "CASE é a utilização de meios de suporte baseados em computador no processo de desenvolvimento de software".

No âmbito deste livro, definimos **CASE** como um conjunto de técnicas e ferramentas informáticas que auxiliam o engenheiro de software no desenvolvimento de aplicações, com o objectivo de diminuir o respectivo esforço e complexidade, de melhorar o controle do projecto, de aplicar sistematicamente um processo uniformizado e de automatizar algumas actividades, nomeadamente a verificação da consistência e qualidade do produto final e a geração de artefactos. Uma ferramenta CASE não é mais do que um produto informático destinado a suportar uma ou mais actividades de engenharia de software, relacionadas com uma (ou mais) metodologia(s) de desenvolvimento.

Tendo em conta esta definição, e o âmbito da Engenharia de Software referido no Capítulo 1, devemos incluir no universo das ferramentas CASE aplicações com funcionalidades tradicionalmente não incluídas. Um bom exemplo disto são ferramentas que auxiliam a gestão de projectos. No entanto, e dado que o foco do livro é o UML, iremos concentrar a análise a efectuar nas ferramentas cujas funcionalidades poderão ser cobertas pelo UML, que tipicamente são as da área de modelação. Esta análise será concretizada nos capítulos 13 e 14, com o estudo de duas ferramentas específicas.

Um dos principais objectivos que há muito tempo se procura atingir com estas ferramentas é a implementação de um ambiente integrado que permita a aplicação de uma abordagem *concept to code* (isto é, "desde a concepção até à implementação") para o desenvolvimento de sistemas de informação. No entanto, este objectivo foi frequentemente comprometido por diversas razões. Uma das mais relevantes tem a ver com a incapacidade de suportar, de forma integrada, todas as activi-

dades das várias fases do processo, e sobretudo de automatizar várias delas (nomeadamente a geração automática de código).

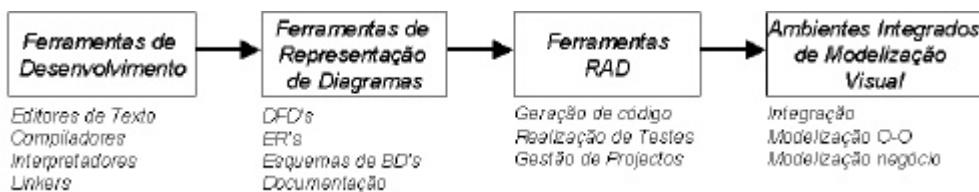
As alterações significativas ao nível da tecnologia, dos métodos de análise e desenho, e a crescente preocupação com a modelação do negócio ao mais alto nível, são algumas das razões que vieram trazer preocupações acrescidas e reforçar a necessidade da implementação do conceito *concept to code*.

Os estudos existentes no mercado não são consistentes sobre as vantagens da utilização deste tipo de aplicações. Se alguns ([Banker91], [Finlay94], [Iivari96]) apontam para aumentos de produtividade com a introdução de produtos CASE, outros ([Orlikowski93], [Vessey92]) chegam à conclusão de que estes benefícios são difíceis de atingir e quantificar.

12.2 Evolução Histórica

Em termos históricos, desde muito cedo que se tornou evidente a necessidade da utilização de ferramentas para auxilio do programador no desenvolvimento de software. Por muito elementares e primitivas que fossem, já as primeiras aplicações necessitavam de suporte de outras ferramentas para serem desenvolvidas e executadas. Numa primeira fase, estamos a falar de ferramentas onde se incluíam os tradutores, compiladores, assembladores, pré-processadores, *linkers* e *loaders*. Numa segunda fase, algumas evoluções tecnológicas, entre as quais se destaca a possibilidade da partilha do tempo de computação, levaram ao desenvolvimento de outras ferramentas que complementaram as anteriores, tais como os editores de texto, *debuggers*, verificadores de código e software para controle de versões.

No início da década de 70, surgiu no mercado o sistema operativo UNIX e respectivos utilitários, que são frequentemente apontados como um dos primeiros conjuntos de ferramentas integradas de apoio ao desenvolvimento. Apesar da sua simplicidade, este conjunto de utilitários disponibilizava um ambiente integrado e uniforme, simples de utilizar por um técnico informático. No entanto, alguns teóricos mais puristas teriam alguma dificuldade em classificar o conjunto destes utilitários como uma ferramenta CASE.

Apenas no início da década de 80 é que surgem no mercado as primeiras ferramentas que se consideram actualmente como integrando o universo CASE. O Excelerator, uma das primeiras ferramentas CASE unanimemente considerada como tal, surgiu em 1984. A crescente importância que foram tendo no processo de desenvolvimento está directamente relacionada com um conjunto de factores decisivos que contribuíram para o crescente sentimento da necessidade deste tipo de ferramentas:

- A mudança do ênfase das actividades de programação para actividades de análise e desenho de software, de modo a possibilitar a ultrapassagem dos diversos problemas que se reconheciam aos métodos de trabalho ad-hoc (ver Secção 3.3).
- Utilização de computadores pessoais e de interfaces de trabalho gráficas.
- O aparecimento de diversas técnicas de modelação de sistemas, que implicavam o desenho de diagramas gráficos (tais como os fluxogramas ou diagramas de fluxos de dados), em que a representação destas notações em papel, ou em ambientes orientados ao caracter, se tornava impraticável à medida que a respectiva complexidade aumentava (quaisquer correcções que fossem necessárias implicavam sempre refazer tudo de novo).
- O aumento da complexidade e do tamanho do software, associado às maiores capacidades do hardware.

As ferramentas desta fase tiveram essencialmente três grandes preocupações: ajudar na elaboração da documentação, na produção de diagramas e no suporte das actividades de análise e desenho.

De realçar que por esta altura começaram a surgir ferramentas que suportavam técnicas e notações apenas de uma única metodologia, enquanto outras mais abrangentes, incluíam suporte para diferentes notações. No primeiro caso, incluem-se sobretudo ferramentas das grandes consultoras internacionais.

Durante a década de 80, assistimos à crescente incorporação de funcionalidades automáticas, nomeadamente a verificação da consistência entre modelos e a geração de modelos de desenho a partir de modelos de análise. Em finais dos anos 80, surgiram as primeiras fer-

ramentas de geração automática de código, a partir das especificações de desenho, e com suporte ao trabalho em equipa. Data também desta altura o aparecimento de ferramentas de outras áreas e que nós consideramos integradas no universo das ferramentas CASE, tais como ferramentas para gestão de projectos, elaboração de estimativas ou suporte à realização de testes.

Figura 12.2: Evolução das ferramentas de apoio ao desenvolvimento de software.

No início dos anos 90, muitas das ferramentas CASE passaram a ser designadas por ferramentas RAD (*Rapid Application Development*), o que traduzia a preocupação de aumentar o ritmo do desenvolvimento aplicacional. Exemplos destas ferramentas foram (e algumas ainda são) o Visual Basic, Sql Windows, Dephi, Powerbuilder, Oracle Designer e Oracle Developer, que estão frequentemente vocacionadas para auxiliar o desenvolvimento de software para ambientes cliente-servidor.

A partir de meados dos anos 90, com a crescente importância das abordagens orientadas por objectos e o desenvolvimento de componentes, a terminologia utilizada por alguns autores passou a ser ferramentas de modelação visual. No entanto, a expressão CASE permanece, no nosso entender, como a mais abrangente de todas.

A evolução dos conceitos e ferramentas CASE nem sempre foi um caminho simples e bem sucedido. Por exemplo, a iniciativa da IBM designada por *AD/Cycle* fracassou em 1992. Esta iniciativa tinha por objectivo definir um conjunto de especificações standard, que permitissem a todas as ferramentas de desenvolvimento comunicar entre si. Igualmente, a grande maioria dos primeiros fabricantes deste tipo de ferramentas já não existe ou foi adquirida por outras empresas.

Mais recentemente, a introdução dos conceitos da orientação por objectos veio de alguma forma revolucionar o mercado, quer porque

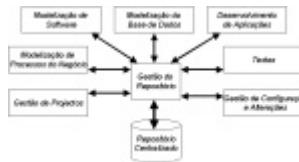
uma parte significativa das ferramentas tradicionais teve que se "reinventar", e incorporar novas técnicas de modelação integradas (ou não) com as abordagens estruturadas já existentes (é o caso do System Architect da Popkin Software); quer porque surgiram no mercado novas ferramentas que suportam exclusivamente este paradigma (é o caso da ferramenta Rose da Rational). Neste contexto, assume particular destaque o UML, que vem assumindo um papel crescente ao nível das notações de modelação. Hoje em dia, praticamente todas as ferramentas que detêm uma quota de mercado mais significativa incorporam algum suporte para o UML.

Esta é uma das "áreas quentes" das ferramentas CASE que se comcentram na modelação de software: a dúvida sobre a preponderância de determinadas notações e consequente implementação pelas ferramentas. A nossa opinião é que as metodologias e notações orientadas por objectos, e em particular o UML, acabarão por se tornar preponderantes no futuro, ao nível da especificação de software.

Outras preocupações importantes são a inclusão de funcionalidades de modelação de processos de negócio, ou a disponibilização de ambientes de programação integrados com as ferramentas de modelação. Esta última fase marca também a crescente preocupação com o desenvolvimento de funcionalidades de apoio ao trabalho em equipa e à gestão de projectos. Esta variedade de funcionalidades e de ferramentas coloca outros desafios adicionais, tais como os aspectos de extensibilidade e de abertura ou mecanismos de integração (ver Secção 9.7 a propósito do XMI).

A tendência para a aquisição e fusão de empresas que actuam na área de modelação, e que já vem desde há alguns anos, continuará. Uma das sequências mais curiosa foi a que culminou na aquisição da Sterling Software pela Computer Associates em 2000, e que teve vários passos intermédios:

- Em 1996, dois dos fabricantes mais relevantes de produtos CASE da altura, a Cadre e a Bachman, juntaram-se numa nova companhia designada por Cayenne.
- A Cayenne foi por sua vez adquirida pela Sterling Software em 1998.

- A Sterling Software já tinha adquirido a Knowledgeware em 1994.
- Também em 2000, a Computer Associates adquiriu a Platinum.
- A Platinum já tinha anteriormente adquirido a LogicWorks.

Todas estes fabricantes disponibilizavam à data da sua aquisição produtos relevantes no mercado, que são actualmente todos comercializados pela Computer Associates.

12.3 Arquitectura das Ferramentas CASE

A maioria das ferramentas CASE especializa-se sobretudo numa tarefa específica do processo de desenvolvimento de software. Algumas concentram-se na disponibilização de funcionalidades relevantes para a fase de concepção (por exemplo, elaboração de diversos diagramas), enquanto outras estão particularmente direccionadas para a fase de implementação (por exemplo, desenvolvimento visual, geração de código e apoio à realização de testes).

Seguindo uma estratégia *best-of-breed*, que significa seleccionar a melhor ferramenta para cada funcionalidade, e dada a abrangência da definição de CASE, estas ferramentas não têm necessariamente que pertencer todas ao mesmo fornecedor. Por isso, tornou-se necessário desenvolver mecanismos que facilitassem a integração e partilha de informação entre elas, uma vez que a mesma informação pode ser relevante para mais do que uma ferramenta. Um bom exemplo disso é a informação sobre os componentes de uma aplicação, que é produzida por uma ferramenta de análise e desenho, e que é essencial para um utilitário que tenha como objectivo parametrizar e executar testes de forma automatizada.

A arquitectura típica das ferramentas CASE (ou pelo menos uma que se considera mais adequada para este tipo de aplicações) é constituída por um conjunto de aplicações / componentes, suportados por um repositório integrado, como se representa na Figura 12.3.

Figura 12.3: Arquitectura genérica das ferramentas CASE.

Repositório

O termo **repositório** designa o componente da arquitectura das ferramentas CASE que é utilizado como meio de armazenamento, gestão e partilha de objectos, modelos, documentos, ou quaisquer outros artefactos, produzidos por algum dos restantes componentes que completam a arquitectura.

Na prática, o papel do repositório pode ser concretizado através de uma base de dados, como é o caso típico dos fornecedores que possuem simultaneamente produtos CASE e bases de dados (casos da Oracle e da Sybase), mas muitos produtos utilizam um simples sistema de gestão de ficheiros, alguns com formatos proprietários.

O repositório de uma ferramenta CASE é particularmente relevante, uma vez que facilita a gestão de modelos elaborados, e a respectiva reutilização, disponibilizando para isso mecanismos potentes de pesquisa. Tipicamente, o seu conteúdo incluirá:

- *Templates* de âmbito variado, nomeadamente de diagramas e de documentos, que facilitam a elaboração de artefactos concretos a partir de modelos genéricos.
- *Templates* e *frameworks* aplicacionais, a partir dos quais podem ser construídos "esqueletos" de aplicações em função de um conjunto de parâmetros.
- Bibliotecas de objectos, classes e componentes, que para além de eventuais componentes que possam vir inicialmente com as ferramentas, permitem a integração de outros desenvolvidos ao longo do tempo.
- Diagramas diversos que resultam da modelação do sistema.
- Código fonte, programas executáveis e aplicações empacotadas prontas para distribuir aos utilizadores finais.

- Ficheiros de dados para testes e *scripts* de execução dos mesmos.

O repositório apresenta as funcionalidades típicas de um sistema de gestão de bases de dados, nomeadamente no que diz respeito a:

- Garantia de integridade de dados.
- Partilha de informação.
- Suporte ao trabalho concorrente de vários utilizadores.
- Facilidades de realização de operações de pesquisa.

O repositório é um componente crítico ao providenciar mecanismos e estruturas de dados para a integração entre as ferramentas, constituindo-se como o meio de armazenamento comum, para todos os artefactos. No entanto, e de modo a garantir o sucesso do repositório, enquanto facilitador da partilha de informação, é necessário que sejam definidos adicionalmente os seguintes aspectos:

- Um formato comum para troca de informação descritiva dos artefactos.
- Uma interface comum para aceder e utilizar os artefactos.

12.4 Mecanismos de Integração entre Ferramentas

Para além do repositório (ou complementarmente), existem outras possibilidades para realizar a integração entre ferramentas CASE. A alternativa mais simplista de partilha de informação entre duas ferramentas distintas consiste na exportação e importação de dados utilizando formatos relativamente universais (por exemplo, campos de texto separados por vírgulas, segundo formato tipo CSV). Esta alternativa, no entanto, não garante qualquer integridade da informação, uma vez que o mesmo conceito se encontra potencialmente duplicado, não sendo possível garantir a coerência da informação: quaisquer alterações numa das cópias não são reflectidas nas restantes.

Outra alternativa tem a ver com a construção de componentes partilhados, em linguagens como C++, Java ou Visual Basic, que podem ser utilizados por várias ferramentas para acesso a funcionalidades comuns. Por exemplo, a funcionalidade de pesquisa de objectos e a implementação de mecanismos de suporte ao trabalho de vários

utilizadores, são relevantes para várias ferramentas, e podem ser implementadas por um componente comum.

Uma vez que, hoje em dia, proliferam ferramentas desenvolvidas por diversos fabricantes, à partida não integráveis, um dos mecanismos mais interessantes para facilitar a integração entre ferramentas é a utilização de um repositório "independente", que facilmente suporta o armazenamento de diversos componentes; um bom exemplo disso foi o produto que a Microsoft desenvolveu com o nome de Repository, e que actualmente se encontra integrado no SQL Server 2000 Meta Data Services. Esta neutralidade, recorrendo a *standards* de armazenamento e acesso aos objectos, permite aos fabricantes utilizarem esta tecnologia para o desenvolvimento e reutilização de componentes *best-of-breed*, e aos clientes obterem o melhor dos dois mundos: (1) as melhores ferramentas, (2) integradas entre si. Os Meta Data Services da Microsoft incluem:

- Um modelo de descrições partilháveis e reutilizáveis, Open Information Model, com a definição de diversos tipos e relações, e que se encontra descrito em UML.
- Um conjunto de interfaces uniformizadas, para acesso a metadados, através de COM e XML, e que pode ser utilizado para definir modelos de informação.
- Um sistema de gestão de objectos do repositório, em que interfaces COM e SQL podem ser utilizadas para aceder aos objectos armazenados em SQL Server.

O *Software Development Kit,* versão 3.0, relativo aos Meta Data Services está disponível para *download* num dos sites da Microsoft (*msdn.microsoft.com/downloads*).

O *CASE Data Interchange Format* (CDIF) é outra iniciativa desenvolvida nesta área, conduzida pela Electronic Industries Association. O CDIF começou a ser desenvolvido em 1987, tendo sido originalmente publicado em 1991 e revisto em 1994 e em 1997, de forma a incorporar contributos posteriores. O CDIF consiste numa família de standards, que definem: (1) uma arquitectura única para a troca de informação entre ferramentas de modelação de diferentes fabricantes, e (2) as interfaces entre os componentes que implementam esta arquitectura.

(Para mais informação sobre esta iniciativa consulte-se *www.eigroup.org/cdif*)

Durante a década de 90 muitas ferramentas aderiram a este standard, mas com a crescente utilização das notações orientadas por objectos, e em particular com o UML, perdeu um pouco da importância (e do sucesso) que lhe era atribuída inicialmente. Para isso muito contribuiu o desenvolvimento do XMI, já referido na Secção 9.7, como novo standard de integração entre ferramentas que suportam UML, e baseado em XML.

12.5 Taxonomia das Ferramentas CASE

A inclusão de um produto no universo das ferramentas CASE depende da abrangência da definição considerada. Por exemplo, uma parte significativa dos informáticos teria dificuldade em considerar um "simples" editor de texto como uma ferramenta CASE. Normalmente, pensamos sempre em ferramentas de suporte a actividades mais "nobres", e a edição de código não traz aparentemente qualquer valor acrescentado ao desenvolvimento de software. No entanto, se pensarmos no significado exacto do termo, e que sem um editor de texto não seria possível produzir um programa (para além de que ele pode mesmo automatizar algumas tarefas, como sejam a substituição de palavras), seremos levados a reconsiderar e a inclui-lo nesta classificação.

Os critérios utilizados para classificar as ferramentas CASE são muito diversos. Os mais significativos incluem: (1) a análise das funcionalidades disponíveis; (2) o papel que representam para os gestores ou para elementos técnicos; (3) a possibilidade de serem utilizados nas várias fases do processo de desenvolvimento de software.

Tendo em conta a proliferação de aplicações nesta área, uma taxonomia das ferramentas CASE é particularmente importante, pois facilita a compreensão da abrangência de uma determinada ferramenta e da sua aplicabilidade nas fases e actividades do processo de desenvolvimento de software. Para além destes questões, a classificação destas ferramentas facilita ainda a realização de análises comparativas.

Uma primeira classificação das ferramentas CASE pode ser efectuada com base nos seguintes critérios:

- Fases do processo de desenvolvimento às quais as ferramentas se aplicam:
  - Ferramentas *Upper-Case* são aplicações que se especializaram na fase de concepção do software (ferramentas de análise e especificação e/ou modelação de requisitos).
  - Ferramentas *Lower-Case* são aplicações utilizadas na fase de implementação (ferramentas de desenho técnico, de edição e compilação de código e de testes).
- Utilização das ferramentas em actividades especificas de uma fase/tarefa ou concebidas para actividades que se desenrolam ao longo de todo o ciclo: um exemplo típico das primeiras são as ferramentas que implementam ambientes de desenvolvimento, enquanto que nas segundas se pode incluir uma ferramenta de gestão de projectos.

Uma outra classificação mais detalhada agrupa as ferramentas nas seguintes categorias:

- **Modelação de processos de negócio**: ferramentas orientadas para a análise e especificação dos processos de negócio, que permitem verificar como os objectivos estratégicos de negócio são concretizados em processos. Para além de utilizarem diversas notações e diagramas para a representação de informação do negócio (cadeia de valor, responsabilidades e funções da organização), recorrem com frequência a técnicas de simulação e análise de custos (por exemplo, análise ABC). Estas funcionalidades possibilitam a utilização destas ferramentas para compreender o funcionamento da empresa e identificar problemas ou oportunidades de melhoria (e não para o desenvolvimento de software). Esta categoria não é por vezes considerada como integrando o universo das ferramentas CASE. Exemplos: Aris Toolset (*www.ids-scheer.com*), Mega Suite (*www.mega.com*), Provision (*www.proformacorp.com*).

- **Modelação de análise e desenho do sistema**: esta é a categoria onde se pode incluir a maioria das ferramentas CASE. Permitem normalmente relacionar modelos de processos com os modelos e requisitos a implementar nos sistemas de informação, recorrendo a técnicas referidas no Capítulo 3. É nesta área que o paradigma da orientação por objectos e o UML maior impacto têm tido. Como exemplos desta categoria de ferramentas, que suportam o paradigma da orientação por objectos, temos o Rose (*www.rational.com*), o Paradigm Plus (*www.cai.com*), o GDPro (*www.advancedsw.com*). Se considerarmos as ferramentas que adicionalmente suportam abordagens estruturadas temos o System Architect (*www.popkin.com*), o PowerDesigner (*www.sybase.com*) e o Silverrun (*www.silverrun.com*).
- **Desenho de bases de dados**: aparecem na sequência das ferramentas anteriores (muitas vezes de forma integrada), mas especializaram-se na definição lógica e física da estrutura das bases de dados. Exemplos: System Architect (*www.popkin.com*) o PowerDesigner (*www.sybase.com*) e o Erwin (*www.cai.com*).
- **Programação de aplicações**: ferramentas que normalmente incluem num ambiente único e integrado (se bem que as mais antigas podem ainda funcionar de forma autónoma) funcionalidades de edição de programas, de concepção da interface, e outros programas tais como os interpretadores, compiladores, geradores de código e *debuggers*. Exemplos: Visual Basic e Visual C++ (*www.microsoft.com*), Delphi (*www.borland.com*) e Powerbuilder (*www.sybase.com*).
- **Gestão de alterações no software**: suportam o trabalho em equipa, e implementam funcionalidades de gestão de versões, de mecanismos de *check-in* e *check-out* (operações que garantem que apenas uma pessoa acede com permissões de alteração a um determinado artefacto do sistema), de gestão da configuração e distribuição do software. Exemplos: Visual Sourcesafe (*www.microsoft.com*) e ClearQuest (*www.rational.com*).
- **Testes**: esta categoria compreende ferramentas que permitem a definição de regras de testes, a geração de *scripts* para posterior execução de testes, a definição de dados para testes, o controle e a gestão de erros, e a obtenção de estatísticas relacionadas com esta informação. Exemplos: Suite TestStudio (*www.rational.com*) e TestWorks (*www.soft.com*).

- **Orientadas para a Gestão de Projectos**: são ferramentas cujas principais funcionalidades se destinam a facilitar as tarefas de gestão e coordenação dos projectos, com recurso a técnicas tais como o planeamento e estimativa de tempos, custos e recursos, a utilização e afectação de recursos ao projecto, definição de responsabilidades. Por vezes, incluem ainda facilidades de auxílio na aplicação de uma metodologia de desenvolvimento de software (muitas vezes disponibilizando informação sobre melhores práticas, casos de estudo, e uma *knowledge base* sobre todo o processo). Exemplos: Project (*www.microsoft.com*) e Juggler (*www.cse.dcu.ie/catalyst*).

12.6 Vantagens e Problemas das Ferramentas CASE

A introdução de ferramentas CASE numa organização pressupõe uma predisposição para a aplicação de regras e princípios a todo o processo de desenvolvimento, sendo esta pré-condição já de si um aspecto positivo no processo de melhoria do desenvolvimento de software numa organização. Podemos identificar algumas das principais vantagens que resultam da aplicação deste tipo de ferramentas:

- **Uniformização** do processo de desenvolvimento, das actividades realizadas, e dos artefactos produzidos.
- **Reutilização** de vários artefactos ao longo do mesmo projecto, e entre projectos, promovendo o consequente aumento da produtividade.
- **Automatização** de actividades, com particular destaque ao nível da geração de código e de documentação.
- **Diminuição do tempo de desenvolvimento**, recorrendo à geração automática de diversos artefactos do projecto, ou à reutilização de outros previamente existentes.
- **Integração** de artefactos produzidos em diferentes fases do ciclo de desenvolvimento de software, em que os *outputs* de uma ferramenta são utilizados como *inputs* de outra. Um bom exemplo é a possibilidade de um diagrama de classes originar o esquema relacional de uma base de dados.

- **Demonstração da consistência** entre os diversos modelos e possibilidade de verificar a correcção do software.
- **Qualidade** do produto final superior, pois a utilização de ferramentas impõe um rigor que obriga a uma abordagem mais estruturada no processo de desenvolvimento.

No entanto, nem tudo são aspectos positivos, e estas vantagens podem mesmo ser contrariadas por alguns problemas. No passado, verificou-se frequentemente que os resultados obtidos não estavam de acordo com as elevadas expectativas criadas, o que provocou o fracasso da introdução das ferramentas CASE nas organizações. Um dos factores mais referidos como responsável por esta situação é o elevado tempo de aprendizagem, por vezes requerido para tirar o melhor partido destas ferramentas, e que não é compatível com as exigências das organizações apresentarem resultados o mais rapidamente possível.

Outros problemas detectados incluem a impossibilidade de, numa abordagem mais estratégica, mapear os processos de negócio (modelados em notações facilmente compreensíveis pelos utilizadores) em requisitos de informação. A integração entre o desenho lógico de uma base de dados e a respectiva estrutura física é também uma das áreas que ainda é necessário melhorar, pois é frequente que as ferramentas apenas permitam a definição de um esquema da base de dados, sem ter em conta a respectiva estrutura e organização nos suportes físicos (esta questão é ainda mais relevante no caso das ferramentas que suportam UML, como veremos nos capítulos seguintes). Uma área que ainda hoje constitui um dos principais mitos do software é a da geração automática de código: desde há muito tempo tem sido um dos principais objectivos que se procura atingir com as ferramentas CASE, mas onde ainda não foram obtidos resultados completamente satisfatórios.

12.7 Funcionalidades das Ferramentas CASE

A estratégia de introdução das ferramentas CASE numa organização pode ser diversa, nomeadamente:

- *Suite*: selecção de um conjunto integrado de ferramentas, todas do mesmo fornecedor.

- *Best-of-breed*: selecção das melhores ferramentas para cada funcionalidade, suportadas por um repositório integrado.
- Pontual: selecção de ferramentas para cobrir áreas pontuais.

Existem algumas funcionalidades que a maioria das ferramentas disponibiliza, independentemente da sua área de intervenção, e que estão relacionadas com a gestão e controle do acesso à informação:

- Definição de grupos e de perfis de utilizadores.
- Possibilidade de manter um registo de todas as alterações efectuadas, associado ao controle de versões e à disponibilização de mecanismos de *check-in* e *check-out*.
- Suporte ao trabalho de equipas e a um ambiente multi-utilizador.
- Possibilidade de implementar a noção de projecto e reutilização de artefactos de outros projectos já realizados.

No entanto, é natural que as funcionalidades mais significativas de cada ferramenta sejam especializadas consoante a sua área de aplicação. A título de exemplo, poderemos referir que uma ferramenta direccionada para as actividades relacionadas com gestão de projectos deverá apresentar, entre outras, as seguintes funcionalidades:

- Possibilidade de representar as noções de fase, tarefa e actividade que são executadas ao longo de um projecto.
- Representação de diversos diagramas típicos da gestão de projectos (diagramas de Pert, Gantt, matrizes de alocação de recursos).
- Possibilidade de comparar o esforço planeado com o efectivamente realizado.
- Possibilidade de atribuir actividades a recursos, e analisar a alocação de cada um dos recursos.
- Possibilidade do responsável pela execução de uma actividade introduzir informação sobre o trabalho que foi efectuado.
- Possibilidade de utilizar dados históricos para elaborar estimativas para um novo projecto.
- Possibilidade de analisar não apenas as questões de prazos, mas também as financeiras.

Uma vez que considerámos uma definição muito abrangente do conceito CASE, seria necessário elaborar uma lista específica para cada tipo de ferramenta. Esta descrição é de tal modo exaustiva que sai

fora do âmbito deste livro, até porque o seu objectivo é analisar aquelas que estão relacionadas com a utilização do UML.

Por isso vamos concentrar a nossa análise nas características de natureza funcional relacionadas com as ferramentas de modelação. Optámos pela apresentação de uma lista de funcionalidades, tipo *check-list*, que pode funcionar como um conjunto de critérios de avaliação e selecção das diversas ferramentas de modelação, e como uma tentativa de sensibilizar o leitor para as reais capacidades que pode e deve esperar deste tipo de ferramentas. Esta lista de funcionalidades servirá nos próximos dois capítulos para apresentar e discutir algumas características das duas ferramentas em análise.

A lista de funcionalidades que se segue não é de modo algum exaustiva, nem tão pouco estática. A evolução dos conceitos e da própria tecnologia implicará com certeza a introdução e/ou eliminação de novas funcionalidades. Esta lista encontra-se organizada por grandes grupos de funcionalidades, de forma a facilitar a sua compreensão.

Critérios de Modelação

- Suporte a um ou mais métodos ou paradigmas metodológicos. Neste caso, o suporte à linguagem UML deve cada vez mais ser considerado um requisito fundamental.
- Ferramenta orientada para a aplicação de técnicas de modelação ou de uma metodologia completa. Neste caso, interessa determinar a eventual capacidade de customização da mesma.
- Tipos de modelação suportados: modelação de dados, diagramas funcionais, modelação em UML, modelação do negócio, modelação de instalação/distribuição de componentes.
- Nível de cobertura às várias tarefas do processo de desenvolvimento.
- Separação e integração entre modelação de nível conceptual (conceitos do negócio e de análise) e de implementação (desenho e programação).

- Integração de diversos modelos e capacidade de definir sub-modelos, possibilitando a existência de diversos níveis de abstracção.
- Integração e sincronização entre modelos e interfaces, suportando a devida rastreabilidade (*traceability*).
- Rastreabilidade dos artefactos ao longo de todo o processo (por exemplo, desde o conceito de negócio até à tabela da base de dados, passando pela entidade da análise).
- Possibilidade de converter um modelo entre notações distintas.
- Possibilidade de estender as representações gráficas, quer ao nível dos conceitos quer do aspecto visual (por exemplo, suporte à definição de estereótipos).
- Automatização de actividades de produção de modelos a partir de outros existentes.
- Verificação da consistência entre modelos.

Repositório

- Tecnologia de implementação do repositório: produto comercial disponível no mercado, base de dados proprietária, ficheiro de texto.
- Nível de especialização do repositório, podendo suportar apenas uma ferramenta ou possibilitar o acesso por várias ferramentas de modelação.
- Estrutura do repositório aberta e conhecida, acessível através de uma interface (SQL, COM, XML).
- Possibilidade de estender a estrutura do repositório.
- Facilidade de administração de modo a garantir a segurança do repositório.
- Suporte a versões.
- Suporte ao trabalho em equipa.

Geração de Código

- Possibilidade de análise e verificação da estrutura de um programa.
- Possibilidade de automatizar a produção do código.
- Linguagens de programação suportadas para geração de código (por exemplo, C++, Java, Visual Basic).

- Suporte para o *reverse-engineering* de aplicações existentes.
- Suporte para o *forward-engineering* (geração de código).
- Suporta desenvolvimento cíclico de código (*round-trip engineering*), com introdução de marcas e sem perda de informação.
- Possibilidade de integração com ferramentas de gestão de configurações.
- Possibilidade de geração de interface gráfica.
- Geração de documentação de validação do processo.

Gestão de Configuração e das Alterações

- Comparação de alterações entre versões e produção de *scripts* que reproduzam essas alterações.
- Produção de relatórios de análise de impacto das alterações a efectuar.
- Disponibilização de mecanismos de *check-in* e *check-out* de componentes.
- Definição de níveis de segurança e auditoria aos modelos.

Modelização de Dados

- Integração entre os modelos de análise e a estrutura de uma base de dados. Se pensarmos no caso do UML, estamos a falar de diagramas de classes.
- Suporte para *Reverse Engineering* de bases de dados.
- Suporte para geração de outros objectos de uma base de dados (por exemplo, *triggers* e *stored-procedures*).
- Manutenção de integridade referencial.
- Geração de esquemas de bases de dados, sendo particularmente relevante o suporte para o modelo relacional.
- Lista de bases de dados suportadas (Oracle, Sql Server, DB2, Informix, …). É conveniente o suporte de vários dialectos de DDL/SQL, nomeadamente para os sistemas de gestão de bases de dados mais utilizados.

Documentação

- Geração automática de relatórios e de documentação geral de projectos: estatísticas, dicionário de dados, modelos elaborados, inconsistências, rastreabilidade, operações realizadas.
- Possibilidade de publicação dos modelos em interface Web.
- Nível de detalhe que se pode definir para cada atributo/elemento manipulado pela ferramenta (por exemplo, descrição, pré e pós-condições, etc.).
- Suporta geração automática de documentos com base em *templates*.
- Possibilidades de configurar a geração de documentação.

Extensibilidade

- Existência de linguagem de *scripting* e tipo de capacidades providenciadas (por exemplo, para configuração da ferramenta, geração de código, produção de relatórios, verificação da consistência).
- Capacidade de integração com outras ferramentas e processo como tal é realizado.
- Possibilidade de estender a ferramenta ao nível da documentação gerada.
- Possibilidade de introduzir na ferramenta novos conceitos.

Outras Questões Genéricas

- Facilidade de utilização (ajuda, manuais, controle de consistências, relatórios).
- Funcionalidades de importação e exportação.
- Definição de regras de validação.
- Abrangência de conceitos e de componentes suportados, e nível de detalhe de cada um.
- Obrigatoriedade de utilização e aplicação de standards.
- Suporte a modelos de elevada dimensão (por exemplo, o número de processos ou de entidades num diagrama).

12.8 Geração Automática de Artefactos

Como vimos anteriormente, uma das vantagens apontadas às ferramentas CASE é a disponibilização de diversos mecanismos de automatização de tarefas. Neste aspecto, são particularmente interessantes as funcionalidades que estão relacionadas com a geração de código e de documentação, e que passamos a descrever de seguida.

12.8.1 Round-Trip Engineering

A aplicação do princípio "*concept to code*" significa na prática que deveria ser possível a partir da especificação dos requisitos de um problema, definir e implementar a respectiva solução.

De forma a garantir a correcção e a minimização da intervenção humana, o ideal seria efectuar as fases posteriores de forma automática. A este processo chamamos ***Forward Engineering***. No entanto, em noutros casos é interessante ter a possibilidade de efectuar o percurso inverso, isto é, a partir de código previamente existente, produzir os modelos de desenho e de análise correspondentes. A este ciclo designamos por ***Reverse Engineering***, e ao conjunto dos dois por ***Round-Trip Engineering*** (ver Figura 12.4).

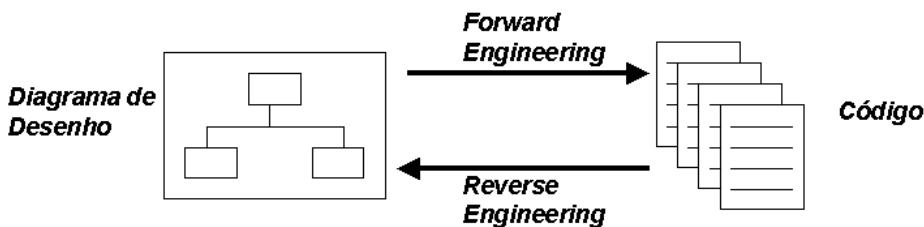



Figura 12.4 : Round-Trip Engineering

Quando os dois conceitos começaram a merecer a atenção da comunidade informática, eram encarados de forma independente, mas actualmente são vistos numa perspectiva integrada.

O *Forward Engineering* tem bastante utilidade da primeira vez que o código é gerado, mas normalmente ocorrem algumas alterações durante o desenvolvimento que são realizadas directamente no código, sem a

correspondente actualização nos modelos. O *Reverse Engineering* permite sincronizar os modelos de desenho e de análise a partir do código fonte.

A funcionalidade de *Reverse Engineering* permite igualmente documentar e clarificar o funcionamento de aplicações antigas (legadas) para as quais não existe qualquer documentação, e/ou quando as pessoas responsáveis pelo seu desenvolvimento já não estão disponíveis. A partir do modelo gerado, poderão ser efectuadas alterações de natureza conceptual que, através de mecanismos de *Forward Engineering,* serão reflectidas novamente no código.

O processo de *Reverse Engineering* consiste na análise de um determinado bloco de código, de modo a capturar a informação relevante para a descrição de um sistema e na sua consequente representação gráfica através de modelos perceptíveis para os engenheiros de software. A partir do *input* de código fonte (ou esquemas da base de dados) são gerados modelos gráficos de análise e desenho, informação sobre a utilização de funções, etc. Estas ferramentas podem ir um pouco mais longe e permitir a reestruturação de código: a partir da análise da sintaxe de um programa, pode ser gerado um fluxograma, que após validação ou alteração, pode gerar um novo programa, desta vez obedecendo às boas práticas da programação. Esta é uma abordagem que pode facilitar a conversão de código legado.

O conceito de *Round-Trip Engineering*, pode ser analisado segundo o comportamento em várias relações:

- Relação entre modelos lógicos de dados e esquemas de bases de dados.
- Relação entre modelos de processos e código de componentes.
- Relação ente diagramas de classes e esqueletos / código de classes ou esquemas das bases de dados.

As ferramentas do primeiro grupo têm tido maior sucesso do que as restantes, uma vez que o número de conceitos das linguagens de programação é muito superior aos conceitos associados à estrutura de uma base de dados. Além disso, o modelo relacional é, actualmente, um standard praticamente universal, enquanto que o número de

linguagens de programação é muito elevado, para que se possa apresentar uma oferta abrangente. Na Figura 12.5 apresenta-se uma concretização do processo de *Round-Trip Engineering*, ao nível de diversos diagramas relacionados com a geração de código.

Figura 12.5: Round-Trip Engineering de Processos

12.8.2 Geração de Documentação

A geração de documentação de uma forma automática é uma funcionalidade importante em qualquer ferramenta CASE. Quando se desenvolve um projecto de raiz, a possibilidade de gerar documentação automaticamente permite libertar a equipa de projecto de uma tarefa demorada e que muitas vezes não chega a ser devidamente concluída.

Por outro lado, existem projectos e aplicações para as quais não existe qualquer documentação. Neste caso, a integração de técnicas de *Reverse Engineering*, poderá possibilitar a geração da documentação necessária. É expectável que as ferramentas CASE disponibilizem um conjunto diverso de funcionalidades de geração de documentação, tais como:

- Documentação de modelos e de código fonte.
- Geração de múltiplos formatos de documentos (Html, Word, etc...).
- Geração de documentação com base em *Templates*.
- Possibilidade de configuração da geração de documentos.
- Possibilidade de personalização e extensão do processo de geração de documentação através de linguagens de *scripting*.

12.9 Avaliação de Ferramentas CASE

Normalmente, a primeira recomendação sobre a selecção de ferramentas CASE seria a utilização de um conjunto limitado de ferramentas, de preferência apenas uma, que integrasse as diversas funcionalidades necessárias, de forma a rentabilizar o investimento e a

garantir, ou pelo menos facilitar, a integração. Contudo, podem existir razões que justifiquem que algumas organizações optem por estratégias diferentes.

Por exemplo, a organização pode ao longo do tempo ter adquirido diversas ferramentas CASE, que actuam em diferentes fases do processo de desenvolvimento (uma ferramenta de modelação, uma de gestão de projectos, outra de gestão de configuração e controle de versões, outra de programação).

Outro exemplo tem a ver com a possibilidade de existir numa organização uma ferramenta com funcionalidades de análise e desenho de software, integradas com a geração de código de modo a facilitar o trabalho dos programadores. Paralelamente, existe um ambiente destinado a utilizadores finais com capacidade de exploração de informação, que utiliza técnicas de quarta geração e que possibilita a construção de relatórios com relativa autonomia. Neste caso, a utilização de uma única ferramenta de "desenvolvimento" de software não faria qualquer sentido.

Os critérios de avaliação são diversos e devem ser aplicados à realidade concreta de uma organização. No entanto, o processo de selecção deve ser semelhante às avaliações de outras aplicações criticas para o negócio de uma organização, e designadamente deve incluir: (1) elaboração de uma grelha de critérios adaptada à realidade da organização, agregados por grandes grupos, e onde deverão estar critérios funcionais, tecnológicos, financeiros e de suporte do fornecedor; (2) atribuição de factores de peso a cada critério, consoante a respectiva importância para a organização; (3) avaliação e classificação com base em informação recolhida e em demonstrações; (4) selecção com base na avaliação efectuada.

Para o sucesso do projecto de selecção é ainda essencial que exista um consenso na organização relativamente à utilização e aos objectivos que se pretendem atingir com a ferramenta: se a organização enfrenta problemas importantes a nível do controle e gestão de projectos, não é com a aquisição de uma ferramenta de automatização de testes que eles serão resolvidos (de facto, também não o serão totalmente com um

programa de gestão de projectos, pois com certeza existirão outros factores a rever).

Os critérios de natureza funcional foram abordados anteriormente. Os critérios de selecção de natureza tecnológica, financeiros e de suporte do fornecedor são genéricos a qualquer selecção de aplicações, e entre vários podemos destacar:

- Financeiros
- Preços das licenças.
- Custos de implementação.
- Custos de manutenção.
- Tecnológicos
- Sistemas operativos suportados.
- Bases de dados sobre as quais trabalham.
- Outros requisitos de hardware (RAM, processador, capacidade em disco).
- Desempenho da ferramenta.
- Escalabilidade.
- Suporte
- Material de suporte disponível (CD, manuais, tutoriais, ficheiros de *help, etc.*).
- Material de apoio traduzido.
- Suporte local disponível.
- Facilidade de utilização.

12.10 Ferramentas de Modelação para UML

Sendo este um livro direccionado para apresentação da linguagem de modelação UML, é natural que abordemos neste capítulo algumas questões relacionadas especificamente com as ferramentas de modelação que utilizam UML. Isto para além de tudo o que já referimos, e aplicável de forma genérica a qualquer tipo de ferramenta.

As ferramentas de modelação para UML podem ser divididas em duas categorias principais, consoante a antiguidade e notações suportadas:

- Ferramentas relativamente recentes, que concentram a sua atenção na modelação de software em UML, ou pelo menos, não realçam o suporte que têm para outras notações. Como exemplos deste grupo temos o Rose da Rational, o GDPro da Advanced Software Technologies e o Paradigm Plus da Computer Associates.
- Ferramentas mais antigas, que tradicionalmente ocuparam posições de destaque no mercado, através da utilização de notações estruturadas, e que gradualmente foram incorporando também o UML, sendo hoje em dia uma das notações que mais destacam. Um bom exemplo desta categoria é o System Architect.

Tipicamente, as ferramentas do primeiro grupo exploram as capacidades do UML até ao limite, incluindo suporte para os aspectos mais avançados do UML (descritos no Capítulo 9). As do segundo grupo são muitas vezes ferramentas de "elaboração de diagramas", recorrendo a diagramas que já suportavam para as áreas em que o UML não é, de base, tão poderoso e completo.

Para além do suporte à elaboração de diagramas, as ferramentas de modelação para UML concentram actualmente a sua preocupação num conjunto de funcionalidades, entre as quais se destacam:

- Mecanismos de *Round-Trip Engineering*, com particular destaque para o suporte de linguagens como Java, C++ ou Visual Basic.
- Produção de documentação, nomeadamente em formatos publicáveis na Internet (HTML).
- Mecanismos de extensibilidade dsiponíveis, recorrendo a linguagens de elaboração de *scripts*.
- Integração entre diagramas. Por exemplo, possibilidade de indicar que as mensagens entre objectos dos diagramas de sequência correspondem a métodos de classes, ou que uma classe num diagrama de classes corresponde a um actor num diagrama de casos de utilização, ou ainda a geração automática de diagramas de colaboração a partir dos diagramas de sequência, e vice-versa.
- Suporte para XMI, o standard da OMG para integração entre ferramentas que suportam UML.

Há algumas áreas, importantes na modelação de software, e que o UML não suporta de base (mas que através de mecanismos de extensão

cada ferramenta pode incluir), e que é importante analisar numa ferramenta que utiliza o UML. Em particular gostaríamos de abordar duas dessas áreas: modelação de bases de dados e modelação do negócio.

12.10.1 Modelação de Bases de Dados

Até à data, o UML não suporta de base a modelação de bases de dados de forma directa. Os conceitos estáticos que o UML representa (pacotes, classes, atributos) podem ser mapeados em conceitos do mundo das bases de dados relacionais (bases de dados, tabelas, colunas), mas são por natureza bastante diferentes. Um dos principais problemas tem a ver com a concretização das diversas relações possíveis entre classes, nas relações entre tabelas (abordaremos mais detalhadamente esta questão nos capítulos seguintes).

No entanto, e como uma das áreas fundamentais de modelação é a geração de esquemas de bases de dados, independentemente da ausência de suporte de base para tal no UML, a ferramenta de modelação deve permitir a integração com funcionalidades de modelação de dados. Para isso, pode seguir uma entre várias alternativas:

- Disponibilizar mecanismos que possibilitem a transformação de um modelo de objectos ou diagrama de classes, num diagrama de dados (diagrama entidade associação), ou mesmo num esquema de manipulação da base de dados. A primeira hipótese é normalmente seguida pelas ferramentas que também possibilitam a aplicação de técnicas de modelação de dados (como o System Architect), enquanto a segunda (a geração do esquema) é a opção das ferramentas que apenas permitem a modelação em UML.
- Exportar metadados sobre o modelo de objectos para uma ferramenta de modelação de dados, que possibilite a sua importação e utilização como base para um modelo de dados.

Se estivermos a falar de um conjunto de ferramentas integradas, é expectável que seja possível manter a sincronização entre modelos de objectos e de dados ao longo de todo o processo de desenvolvimento.

A última versão do Rose (Rose 2001), reconhecendo as limitações ao nível da modelação de dados do UML, veio propor uma extensão ao

UML, designada por *UML Data Modeling Profile*, especificamente concebida para endereçar este problema, e analisada no próximo capítulo.

12.10.2 Modelação do Negócio

Tal como vimos no Capítulo 10, o RUP prevê a aplicação de um conjunto de actividades que se destinam à modelação do negócio. A sua representação e especificação recorre sobretudo a diversos tipos de artefactos documentais e aos diagramas de casos de utilização do negócio. Nesta perspectiva, a modelação de processos de negócio é efectuada em UML através do mesmo diagrama utilizado para a especificação dos requisitos do software: o diagrama de casos de utilização.

No entanto, quando se fala em modelação do negócio pensa-se não apenas em representar graficamente o fluxo de trabalho dentro da organização e as diversas actividades realizadas, mas também em poder associar alguns parâmetros (nomeadamente, o número e o perfil de recursos utilizados, o tempo de duração) de modo a poder identificar onde se situam os problemas, as ineficiências e as oportunidades a explorar, e mesmo a simular o impacto da realização (ou não) de determinadas actividades.

Alguns dos artefactos previstos no RUP podem complementar a notação UML base para fornecer a informação que tradicionalmente se pretende ao nível dos processos de negócio, nomeadamente: a identificação da cadeia de valor; dos produtos e/ou serviços da organização; dos mercados em que actua. No entanto, mesmo este conjunto é insuficiente quando se pretendem avaliar grandezas de natureza quantitativa (tempo, custo e qualidade do produto) e realizar simulações ou análise segundo critérios ABC (*Activity Based Costing*).

As abordagens contabilísticas tradicionais não reflectem a utilização real dos recursos, uma vez que a preocupação está no que se produz e não em como tal acontece. A análise ABC assume que as actividades causam custos porque consomem recursos. Auxilia na análise da eficácia e da eficiência da utilização dos recursos, e na determinação de como as actividades na empresa contribuem para o custo do negócio.

A simulação de processos parte do pressuposto que uma organização resulta de uma série de processos inter-relacionados, e que estes processos consistem em actividades que convertem *inputs* em *outputs*. A simulação de processos de negócio analisa a informação de custos com base nas actividades: a partir da visualização da execução de actividades ao longo do tempo, e das respostas que desencadeiam, permite identificar os problemas e os pontos de estrangulamento existentes, e as oportunidades de melhoria.

A ausência de suporte especifico para técnicas como simulação e análise de custos é um problema partilhado pela maioria das ferramentas comerciais de modelação em UML, uma vez que quase todas foram concebidas para a modelação do software e não dos processos de negócio. No entanto, as características de extensibilidade do UML permitem que esta linguagem seja adaptada na modelação de processos de negócio, utilizando os mecanismos de extensão providenciados (ver Secção 9.3):

Adicionalmente, têm sido propostas algumas extensões ao UML, específicas para modelação do negócio, a mais conhecida das quais talvez seja a proposta por Eriksson e Penker e designada por *Eriksson-Penker Business Extensions* [Penker00]. Estas extensões são já suportadas por algumas ferramentas que as implementam, tal como a Qualiware (*www.qualiware.com*). Outras ferramentas, nomeadamente o Provision da empresa Proforma (*www.proformacorp.com*), adoptam um misto de notação UML, eventualmente estendida, e de outros diagramas mais tradicionais para representar e especificar os processos de negócio.

Uma outra hipótese para solucionar esta lacuna é recorrer aos mecanismos de extensilidade para implementar *scripts* que permitam a criação de interfaces com ferramentas já existentes e especializadas nesta área.

Apesar de tudo, os diagramas de base previstos no UML revelam-se importantes para a modelação dos processos de negócio. Por exemplo, para além dos casos de utilização, os diagramas de actividade e os diagramas de sequência são adequados para a representação do fluxo de trabalho dentro de uma organização; os diagramas de classes e de

objectos podem ser utilizados para representar informação de natureza estática das organizações, como sejam os seus objectivos ou os seus conceitos de negócio e respectivas relações.

12.11 Conclusão

Discutiu-se neste capítulo que a utilização de ferramentas de modelação permite suportar todo o processo de forma mais sistemática, consistente, eficiente e controlável. De facto, parece-nos que, não sendo obrigatória a adopção de tais ferramentas, elas oferecem uma mais valia considerável, principalmente nas tarefas de análise e desenho, quer na realização de novos projectos, quer em projectos de reengenharia, quer ainda como forma de controlo e avaliação de projectos que se encontrem na sua fase de implementação.

Algumas tendências que se têm observado nos últimos anos com as ferramentas CASE são idênticas a outras áreas dos sistemas de informação, e assim no futuro imediato é previsível que se continuem a verificar:

- Consolidação do mercado, quer por aquisições quer pelo desaparecimento de fabricantes economicamente inviáveis.
- Posicionamento das grandes companhias que também têm ofertas nesta área. Particularmente importantes serão os comportamentos da Oracle, da Sybase, Rational e da Computer Associates.
- Crescimento das ferramentas que suportam as abordagens orientadas por objectos, em detrimento das estruturadas, quer por aquisições quer pelo aparecimento de fornecedores especificamente vocacionados para este paradigma.

É nosssa opinião que a crescente utilização das linguagens orientadas por objectos e de ambientes de desenvolvimento de componentes irá com certeza facilitar o crescimento da quota de mercado das ferramentas que utilizam a modelação segundo abordagens orientadas por objectos, e neste particular, a linguagem UML assume papel de destaque.

No final do livro, apresentamos uma lista de ferramentas que consideramos mais relevantes no mercado actual, e um conjunto de

recursos disponíveis na Internet onde o leitor mais interessado poderá consultar alguma informação adicional.

12.12 Exercícios

Ex.87. Qual a importância da definição e aplicação dos standards de integração de ferramentas? Quais as razões pelas quais, na sua opinião, eles têm falhado na realização dos objectivos propostos.

Ex.88. Um dos grandes objectivos há muito tempo perseguido pelas ferramentas CASE é a geração automática de código. No entanto, até à data estas iniciativas não têm tido o impacto esperado. Indique algumas razões pelas quais tal pode ter acontecido.

Ex.89. De que modo a evolução da tecnologia tem condicionado o sucesso das ferramentas CASE?

Ex.90. Segundo a taxonomia de ferramentas CASE apresentada neste capítulo, classifique em termos de importância (fundamental, importante, pouco importante, sem relevância) cada uma das categorias apresentadas, justificando a sua resposta.

Ex.91. Pensa que seria possível utilizar uma ferramenta de modelação de processos, segundo as abordagens tradicionais, para suportar o processo RUP? Justifique a sua resposta.

Capítulo 13 - RATIONAL ROSE

Tópicos

- Introdução
- Interface Gráfica
- Repositório
- Visões e Diagramas UML
- Modelação do Negócio
- Mecanismos de Extensibilidade
- Geração de Código – Caso de Estudo em Visual Basic
- Geração de Modelos de Dados
- Geração da Interface Homem-Máquina
- Geração de Documentação
- Conclusão

13.1 Introdução

O Rose é a ferramenta de modelação visual produzida pela Rational, a mesma empresa que definiu inicialmente a linguagem UML. É de todas as ferramentas que apenas suportam UML (se bem que o Rose suporte ainda outras notações orientadas por objectos) a mais utilizada, sendo considerada internacionalmente como uma referência de grande relevância e notoriedade.

Para a elaboração deste capítulo recorremos à versão Rose 2000 para a plataforma Windows. Durante a elaboração deste livro, saiu uma nova

versão da ferramenta, que não foi possível utilizar. No entanto, existem secções deste capítulo que se referem com maior detalhe a funcionalidades desta nova versão, mas que não chegaram a ser testadas e avaliadas na prática. Para o leitor mais interessado, está disponível uma versão de avaliação na página do fabricante (*www.rational.com*) para *download*.

O Rose faz parte de uma *suite* completa de produtos perfeitamente integrados, que é disponibilizada em várias edições, com o objectivo de suportar diferentes funções associadas ao processo de desenvolvimento de software: analistas, programadores, arquitectos de software e equipas de teste. O objectivo desta *suite* é unificar, optimizar e simplificar o processo de desenvolvimento de software. Entre as várias ferramentas destacam-se: (1) Rational Unified Process, para auxiliar na aplicação da metodologia RUP, (2) RequisitePro, para gestão e controle de requisitos, (3) ClearQuest, para controle de alterações, (4) SoDa, para apoio à elaboração de documentação e (5) Rose, para modelação visual do software utilizando a linguagem UML.

Ao contrário de outras ferramentas nesta área, o Rose foi concebido para facilitar o desenvolvimento de componentes, com suporte nativo para arquitecturas importantes no mercado actual, como sejam *Windows DNA* ou *Enterprise Javabeans*. Neste caso, no início da sessão de trabalho, é possível especificar um *framework*, que inclui um conjunto de elementos de vários modelos previamente elaborados, e que normalmente são necessários para alguns sistemas específicos, possibilitando a sua reutilização pelo utilizador (Figura 13.1).

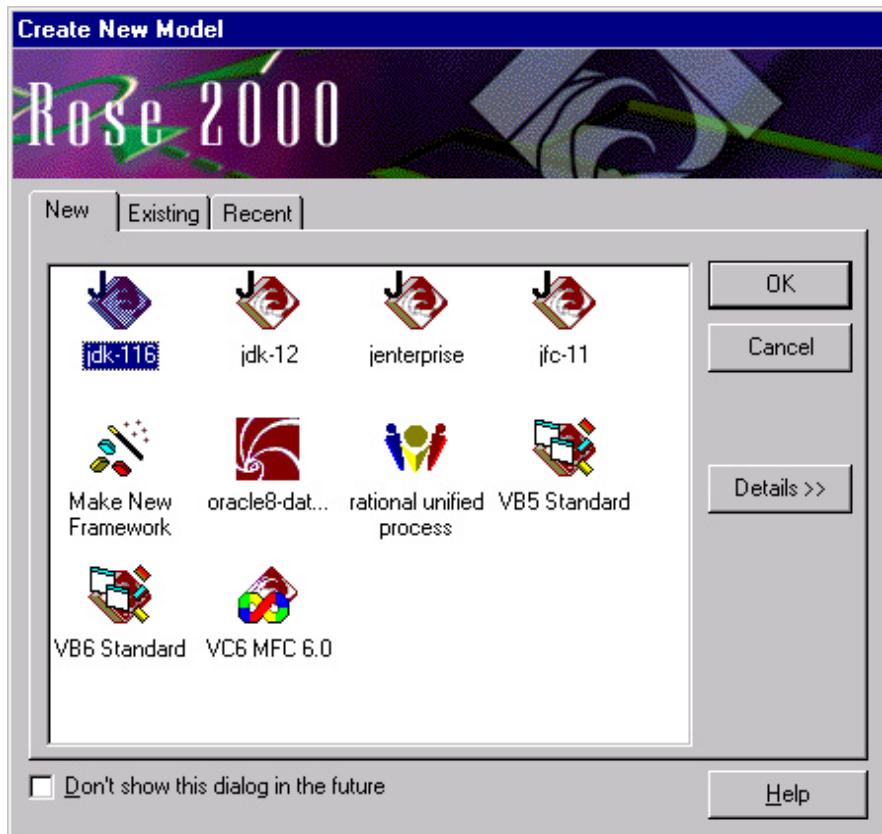



Figura 13.1: Frameworks previamente elaborados no Rose.

Como se pode constatar na figura anterior, está também disponível um *framework* para Oracle8, cuja grande vantagem é incluir logo à partida o conjunto dos tipos de dados dessa base de dados, concretizados sob a forma de classes. Outro exemplo são os *frameworks* para Visual Basic, que adicionam ao projecto referências para as bibliotecas dinâmicas mais utilizadas nesse produto.

O Rose está disponível comercialmente em diversas edições, cujas diferenças se situam ao nível das funcionalidades disponibilizadas, ou das plataformas tecnológicas suportadas (existem versões de Rose distintas para sistemas Windows e Unix). É de salientar ainda a existência de um produto designado por Microsoft Visual Modeler, comercializado em conjunto com outras ferramentas da Microsoft, mas

que é, na realidade, um *subset* do Rose. Este facto representa não só uma estratégia comercial da Rational no sentido de atingir um público alvo mais vasto, mas também é a concretização da estreita relação actualmente existente entre as duas empresas.

O Rose surgiu em 1992 como uma ferramenta de apoio ao desenvolvimento orientado por objectos, suportando nessa altura a modelação de software com base na notação de Booch (que, por essa altura, já estava na Rational). A partir do momento em que o UML foi definido, a ferramenta adoptou-o como a notação preferencial, na qual tem apostado nas suas sucessivas versões.

Tal não significa que exista uma correspondência directa entre as versões do UML e as do Rose. Por exemplo, a primeira versão do Rose com algum suporte para o UML foi a versão 4, que, contudo, não implementava na totalidade a especificação do UML 1.0 já em vigor na altura. A versão Rose 98 suportava o UML 1.1, que já incluía diagramas de actividade, mas que não estavam disponíveis na ferramenta.

A Rational refere a existência de um conjunto de vantagens na utilização do Rose, nomeadamente:

- Integração forte com uma metodologia completa de desenvolvimento de software, de modo a aumentar a produtividade de toda a equipa de desenvolvimento.
- Desenvolvimento baseado no conceito de casos de utilização, resultando numa maior qualidade do software.
- Utilização de uma linguagem de modelação cada vez mais divulgada, o que facilita a comunicação entre elementos de uma equipa de desenvolvimento.
- Funcionalidades de *reverse engineering* que permitem a integração de sistemas previamente existentes.
- Sincronização de modelos e do código ao longo de todo o ciclo de desenvolvimento. Por exemplo, é possível: elaborar um modelo inicial; produzir um protótipo automaticamente a partir do modelo; apresentar o protótipo aos utilizadores; incorporar as sugestões de alteração directamente no protótipo; e finalmente, reflectir essas alterações de forma automática no modelo.

13.2 Interface Gráfica

O aspecto gráfico inicial da ferramenta é muito simples, perfeitamente integrado com os standards do Windows, o que facilita a sua utilização por quem conheça previamente UML, como se pode verificar na Figura 13.2.

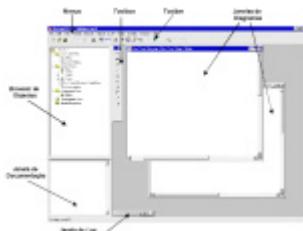



Figura 13.2: Interface gráfica do Rose

Para além dos menus, com aspecto semelhante aos de qualquer produto Windows, e de uma *toolbar* por omissão, facilmente compre-ensível por parte dos utilizadores, inclui ainda as seguintes áreas:

- *Toolbox*, onde se encontram os objectos que podem ser colocados nos diversos diagramas, e que se ajusta automaticamente em função do tipo de diagrama activo.
- *Browser* de objectos, no qual se encontram enumerados todos os artefactos manipulados pelo Rose (nomeadamente, conceitos e diagramas), devidamente hierarquizados. Permite a visualização dos nomes e dos ícones que representam diagramas e elementos do modelo.
- Janela de diagramas, que funciona como uma janela MDI (*Multiple Document Interface*) "dentro da" qual podem ser abertas outras janelas, onde são elaborados os vários diagramas de um modelo.
- Janela de documentação, onde é visualizada e editada a documentação associada a cada elemento de um modelo, sem ter que abrir a respectiva janela de especificação.
- Janela de log, onde podem ser visualizados eventos que são desencadeados pelo Rose, com impacto no repositório.

13.3 Repositório

O Rose armazena todos os artefactos que gere num repositório, podendo nalguns casos incluir referências para documentos externos, que não ficam armazenados fisicamente no repositório. O repositório é concretizado sob a forma de um ficheiro único, designado ficheiro *petal*, acessível directamente pelo sistema operativo, o que coloca algumas limitações, não só à sua capacidade de crescimento, mas sobretudo à segurança da informação armazenada.

Um ficheiro *petal* disponibiliza a funcionalidade de armazenamento permanente de um ou mais elementos do modelo. No caso em que um ficheiro *petal* contem um modelo completo, é normalmente utilizada a extensão `mdl`. Estes ficheiros encontram-se armazenados em formato ASCII, o que facilita a transferência de modelos (completos ou parciais) entre as diferentes versões do produto, mesmo entre plataformas tecnológicas distintas, mas ao mesmo tempo possibilita a realização de alterações directamente no ficheiro, sem qualquer protecção. A estrutura do ficheiro foi elaborada numa lógica de descrição de objectos, enumerando as suas propriedades e os valores assumidos por cada uma.

Adicionalmente, o Rose permite a importação e exportação de modelos para o Microsoft Repository, o que possibilita a partilha de modelos com outras ferramentas de modelação.

No caso em que se utiliza a ferramenta num ambiente de desenvolvimento multi-utilizador, e de modo a facilitar a utilização de diferentes artefactos do modelo por vários utilizadores em simultâneo, o Rose possibilita o armazenamento dos artefactos em ficheiros separados, com extensão diferente. Estes ficheiros ficam armazenados todos na mesma directoria, sendo de destacar os seguintes:

- O ficheiro que contem o núcleo do modelo utiliza a extensão `mdl`.
- Os ficheiros que contêm elementos do modelo controlados e que são packages lógicos utilizam a extensão `cat`.
- Os ficheiros que contêm elementos do modelo controlados e que são packages de componentes utilizam a extensão `sub`.

- O ficheiro que contém o diagrama de distribuição do modelo utiliza a extensão `prc`.
- O ficheiro que contem o conjunto de propriedades do modelo utiliza a extensão `prp`.

No caso dos elementos do modelo não serem unidades controladas, então normalmente utiliza-se a extensão `ptl` para designar o respectivo ficheiro.

De referir que estão disponíveis algumas funcionalidades de apoio ao trabalho em equipa, muitas das quais são suportadas com recurso às funcionalidades do produto Microsoft Sourcesafe, designadamente:

- Possibilidade de controlar versões do modelo, e de comparar alterações efectuadas entre diferentes versões.
- Mecanismos de *check-in* e *check-out*.
- Verificação de quem fez o quê e quando.

13.4 Visões e Diagramas UML

Ao contrário de outras ferramentas de modelação em UML, sobretudo orientadas para a produção de diagramas, o Rose implementa o conceito das visões de um sistema (descrito na Secção 10.4), o qual é fundamental perceber antes de se analisar as técnicas de modelação disponibilizadas. Na prática, este conceito funciona como uma primeira classificação, que permite agrupar todos os artefactos que o Rose gere. O Rose considera a existência de quatro visões de um sistema de software:

- Visão de casos de utilização.
- Visão lógica.
- Visão dos componentes físicos.
- Visão de instalação.

A **visão dos casos de utilização** auxilia na compreensão da utilização do sistema, uma vez que representa as interacções entre os actores e os casos de utilização. Os diagramas utilizados, segundo esta visão, são os diagramas de casos de utilização, os diagramas de sequência,

os diagramas de colaboração e os diagramas de actividade. Tem, por omissão, um diagrama `Main` que pode ser complementado com outros diagramas, de maior detalhe, ao longo de todo o processo.

A **visão lógica** do sistema analisa-o de um ponto de vista estático, e representa a sua concretização em termos de classes e respectivas relações, expressas nos diagramas de classes, nos diagramas de objectos e nos diagramas de estados. Também tem, por omissão, um diagrama `Main`.

A **visão dos componentes** apresenta a organização do software do sistema, incluindo informação sobre o código fonte, executáveis e bibliotecas do sistema. Esta visão apenas inclui os diagramas de componentes, que por omissão também tem um diagrama `Main`.

Por fim, a **visão de instalação** apresenta o modo como é realizada a distribuição de componentes executáveis pelos diferentes nós (plataformas computacionais) de suporte. Apenas inclui o diagrama de instalação, que pode ser útil num ambiente de arquitectura distribuída, em que é possível ter aplicações e servidores em localizações diferentes.

Por conseguinte, e embora sob diferentes visões, o Rose suporta todos os diagramas previstos na linguagem de modelação UML, nomeadamente diagramas de:

- Casos de utilização.
- Classes.
- Colaboração.
- Sequência.
- Actividade.
- Componentes.
- Estados.
- Diagrama de instalação.

Apesar do Rose ser conhecido essencialmente pela utilização da linguagem de modelação UML, estão disponíveis outras notações, nomeadamente a de Booch e a OMT. Para além da contribuição que estas duas abordagens tiveram na definição inicial do UML, a justificação para a inclusão destas abordagens está relacionada com a

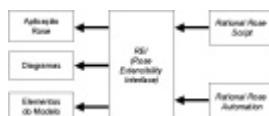
facilitação da conversão, para o UML, de projectos e recursos humanos habituados à utilização destas notações.

13.5 Modelação do Negócio

Nesta área, tal como nas restantes áreas de modelação, o Rose restringe a sua actividade de modelação ao suporte disponibilizado pelo UML, o que significa que não tem nenhuma funcionalidade específica para a representação de processos de negócio que possibilitem a aplicação de técnicas de simulação, de análise de impacto de alterações aos processos de negócio e de identificação de custos segundo perspectivas baseadas na realização de actividades.

Contudo, e através dos mecanismos de extensibilidade que o UML implementa (ver Secção 13.6), é possível representar adequadamente os conceitos de negócio, recorrendo aos diagramas de base do UML, nomeadamente os diagramas de casos de utilização, os diagramas de actividades e os diagramas de sequência para a representação do fluxo de trabalho numa organização e os diagramas de classes para representar qualquer informação de natureza estática das organizações.

Adicionalmente, é ainda possível recorrer aos mecanismos de extensibilidade para implementar *scripts* que permitam a criação de interfaces com ferramentas já existentes e especializadas nesta área.

13.6 Mecanismos de Extensibilidade

A adequação das ferramentas de modelação às necessidades específicas de cada organização é, hoje em dia, virtualmente impossível, em função das inúmeras alternativas que se colocam. Por isso, o Rose permite efectuar a personalização das suas capacidades, de modo a adaptá-las às necessidades específicas de desenvolvimento de uma determinada organização e/ou projecto, através das seguintes possibilidades:

- Personalização de menus do Rose.

- Automatização de funções manuais recorrendo à construção de *scripts.* Estes podem ser utilizados, por exemplo, para a criação de classes e diagramas; actualizações do modelo; geração de documentação.
- Execução de funções do Rose a partir de outras aplicações, recorrendo ao *Rose Automation object* (*RoseApp*).
- Acesso a classes, propriedades e métodos do Rose, directamente no ambiente de desenvolvimento da organização, através de referências à *Rose Extensibility Type Library* nesse ambiente.
- Activação de *add-ins* no Rose, usando o *Add-In Manager*.

Para além de estar concebido para a produção de software baseado em componentes, o Rose é ele próprio baseado em componentes, cuja estrutura é definida pelo modelo *Rose Extensibility Interface* (REI). A Figura 13.3 ilustra a forma como a funcionalidade do Rose pode ser estendida, ao representar os seus diversos componentes:

- **Aplicação Rose**: inclui os objectos que disponibilizam a interface para acesso às funcionalidades internas da aplicação.
- **Interface de Extensibilidade do Rose**: é o conjunto de interfaces partilhadas entre o *Rose Script* e a *Rose Automation* (OLE), para acesso às funcionalidades da aplicação Rose.
- ***Rose Script***: inclui o conjunto de objectos do *Rose Script* que permitem a utilização de *scripts* para a automatização de funcionalidades do Rose.
- ***Rose Automation***: é o conjunto de objectos do *Rose Automation* que permitem que o Rose funcione como controlador ou como servidor OLE.
- **Diagramas**: são os objectos da *Rose Extensibility* que providenciam a interface com os diagramas e visões da aplicação.
- **Elementos do Modelo**: são os objectos da *Rose Extensibility* que providenciam a interface com os elementos dos modelos Rose.

Figura 13.3: Mecanismos de extensibilidade do Rose.

Concretamente, quem quiser utilizar a interface de extensibilidade do Rose (REI) pode recorrer a um conjunto de classes que se encontram agrupadas, e das quais se destacam os seguintes grupos:

- *Application Classes*: classes que realizam operações ao nível do modelo no seu todo (por exemplo, criação de novos modelos ou a selecção de um modelo existente).
- *Use Case Classes*: classes que disponibilizam um conjunto de propriedades e métodos que permitem a definição e manipulação dos diagramas de casos de utilização de um modelo.
- *Model Classes:* classes que providenciam acesso às propriedades e métodos dos objectos de um modelo.
- *Logical Classes*: permitem definir e manipular as classes de um modelo, definir e consultar características e relações de classes e manipular diagramas de classes.
- *Physical Classes*: classes que disponibilizam as propriedades e métodos necessários para se poder criar e manipular módulos.
- *Deployment Classes*: classes que permitem a manipulação das características dos elementos de processamento de um modelo, dos dispositivos físicos sem capacidade de execução, dos fluxos de execução e a representação visual de dispositivos e processadores.

Da mesma forma que quaisquer outras classes, também as incluídas nas categorias acima enumeradas disponibilizam um conjunto de atributos e métodos que podem ser acedidos, a partir de outros programas ou em *scripts* do Rose.

Apresentamos de seguida algumas das funcionalidades de extensibilidade referidas anteriormente, deixando a parte especificamente relacionada com a geração de código e de esquemas de bases de dados para a Secção 13.7.

13.6.1 Extensibilidade dos Menus

A configuração de menus é o mecanismo mais simples disponibilizado pelo Rose para adicionar novas funcionalidades à ferramenta. Podem acrescentar aos menus existentes novas opções, alterando para isso o conteúdo do ficheiro `Rose.mnu`. Estas extensões podem incluir a

definição de novos submenus, ou de opções de menu, que executem (1) *scripts* do Rose; (2) comandos de sistema; ou (3) programas externos. Podem também ser adicionados separadores entre opções dos menus.

Por exemplo, caso se pretendesse desenvolver documentação específica para obter informação do modelo, nomeadamente uma lista com todas as classes ou casos de utilização, poderiam ser acrescentadas duas novas opções ao menu Report (Figura 13.4), que invocariam *scripts* especificamente desenvolvidos para esse efeito.

Figura 13.4: Configuração dos menus do Rose.

Esta alteração dos menus foi produzida através da modificação do código do ficheiro `rose.mnu`, tal como pode ser observado de seguida.

```
Menu Report
{
option "Show &Participants in UC..."

   {
       enable %selected_items:empty:false
       RoseScript $SCRIPT_PATH\participants.ebx
   }
 option "&Documentation Report..."
   {
       RoseScript $SCRIPT_PATH\reportgen.ebx
   }
       Menu "Relatorios Especificos"
       {
              Option "Lista de Classes"
              {
                     RoseScript $SCRIPT_PATH\listaclasses.ebx
              }
              Option "Lista de Casos de Utilização"
              {
                     RoseScript $SCRIPT_PATH\listausecases.ebx
              }
       }
}
```

13.6.2 Scripts no Rose

A linguagem de *scripting* utilizada pelo Rose designa-se por *RoseScript*, e é uma versão estendida da linguagem BasicScript da empresa Summit (informação adicional sobre esta linguagem pode ser encontrada online em *basicscript.summsoft.com*). As extensões de *scripting* do Rose possibilitam a automatização de funções específicas ou a implementação de outras não disponíveis através da interface do Rose.

Figura 13.5: Editor de scripts.

O Rose disponibiliza um editor de *scripts*, integrado no ambiente, que possibilita a criação de novos *scripts* ou a alteração de outros existentes. Para o leitor mais interessado, aconselhamos a consulta do manual de referência do *RoseScript* que acompanha o produto .

Na Figura 13.5 podemos observar o editor de *scripts* integrado no Rose, bem como o aspecto da sintaxe da linguagem, utilizando como exemplo o *script* para a geração de esquemas de bases de dados utilizado pelo Rose.

13.6.3 Rose Automation

A disponibilização de funcionalidades OLE para integração com ferramentas de outros fornecedores é providenciada pelo Rose através do mecanismo *Rose Automation*, o qual oferece duas alternativas:

- **Aplicação Rose actua como cliente OLE**: Através da invocação de um objecto *OLE* num *script* do Rose, e tirando partido do funcionamento do Rose enquanto *automation controller*. Esta funcionalidade permite que o Rose possa executar funções de aplicações Windows, tais como o Word e o Excel, através dos mecanismos típicos de *OLE automation*.
- **Aplicação Rose actua como servidor OLE**: Através da invocação por outra aplicação, que também suporte OLE, de operações disponibilizadas pelo *OLE automation object* do *Rose*. Por exemplo, pode desenvolver-se em Visual Basic um programa que execute o Rose, abra um modelo e obtenha informação, que depois utilize para criar um relatório em formato Excel.

A *Rose Automation* está disponível para qualquer aplicação e linguagem que possa invocar objectos COM, através do conjunto de classes e interfaces da REI, como foi descrito na Secção 13.6.

13.6.4 Rose Add-Ins

Tal como a maioria dos ambientes de desenvolvimento cliente-servidor, também o *Rational Rose* tem a possibilidade de acrescentar extensões ao produto base, através da sua caracterização como componente *add-in* no ambiente do produto. As funcionalidades destes add-ins são acessíveis através de invocações ao objecto *RoseAddInManager*.

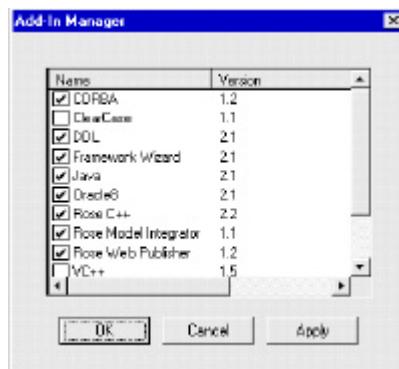



Figura 13.6: Rose Add-in Manager.

No ambiente integrado do *Rose*, existe a possibilidade de indicar quais os *add-ins* que se pretendem utilizar, através de referências no Add-In Manager (Figura 13.6).

Muitos destes *add-ins* são utilizados para suportar processos de *forward* e *reverse engineering*.

13.6.5 Rose Extensibility Type Library

O Rose possibilita a utilização da *Rose Extensibility Interface* (REI) a partir de um ambiente de desenvolvimento, recorrendo a mecanismos de referência de bibliotecas de tipos. Por exemplo, a REI permite que os programadores utilizem no Visual Basic os nomes das classes do Rose, directamente no código que desenvolvem, em vez de referir uma classe genérica. Deste modo, a verificação da sintaxe das propriedades e métodos e a detecção de eventuais erros é efectuada em tempo de compilação (*static binding*) e não no momento de execução do código (*dynamic binding*).

13.7 Geração de Código – Caso de Estudo em Visual Basic

Tal como a maioria das linguagens de modelação, também o UML não especifica à partida regras para a geração de código. Estas podem ser eventualmente implementadas ao nível de cada ferramenta que utilize UML. No Rose, esta funcionalidade é suportada por diversos mecanismos de extensibilidade, nomeadamente *add-ins* e *scripts*.

O Rose disponibiliza funcionalidades de geração de código para as linguagens C++, Java, Corba-IDL e Visual Basic. Para todas elas tem mecanismos que suportam a geração de código (*forward engineering*) e também *reverse engineering*. O suporte a *round-trip engineering* é levado a cabo através de marcas colocadas no código gerado/carregado e de “variáveis” especiais (comentadas no código, mas processadas pela ferramenta). Para além destas linguagens, suportadas de base, é possível desenvolver *scripts* para suportar a geração de código e *reverse engineering* para outras linguagens.

O Rose efectua todo o processo de geração de código a partir do conceito de classe, isto é, com base na especificação de cada classe (atributos, operações e relações), e em função de valores atribuídos a diversas propriedades associadas ao modelo. De modo a exemplificar este processo, iremos utilizar as funcionalidades que o Rose dispõe para facilitar o desenvolvimento em Visual Basic. Estas baseiam-se num *add-in* que permite a modelação, geração e o *reverse-engineering* do código em *Visual Basic*. O processo segue um conjunto de passos sequenciais, que procuraremos demonstrar de seguida:

- Criar na *Logical View* as classes (com os respectivos atributos e operações) e especificar as relações entre elas.
- Criar na *Implementation View* um componente e indicar que este estará associado a um projecto Visual Basic.
- Associar as classes em causa ao componente.
- Parametrizar atributos e operações de cada classe, em função de informação especifica do Visual Basic.
- Criar o projecto no Visual Basic.
- Gerar o código.

Note-se que as regras de geração de código para outras linguagens suportadas são muito semelhantes às que irão ser apresentadas para a linguagem Visual Basic.

13.7.1 Ferramentas Utilizadas

A integração entre o Rose e o Visual Basic é muito forte. Estão disponíveis diversas ferramentas para auxiliar as várias actividades associadas à geração de código e *reverse engineering* para as aplicações em Visual Basic:

- *Component Assignment Tool.*
- *Code Update Tool.*
- *Model Update Tool.*
- *Model Assistant.*
- *Type Library Importer.*

A *Component Assignment Tool* providencia uma interface fácil de utilizar para:

- Criar novos componentes no modelo.
- Atribuir classes a componentes.
- Associar um componente a um projecto Visual Basic.

O *Model Assistant* fornece uma forma alternativa para:

- Definir, no modelo, atributos e operações específicos para a geração de código em Visual Basic.
- Customizar o código a gerar para uma classe.

Através da *Code Update Tool* é possível gerar o código fonte dos componentes de um projecto em Visual Basic a partir da informação contida no modelo e também:

- Visualizar previamente o código a ser gerado para cada classe.
- Especificar detalhadamente o mapeamento entre classes no modelo e o código, através de integração com o *Model Assistant*.
- Manter o modelo e o código sincronizados, à medida que se detecta que alguns elementos do projecto foram alterados no modelo.

Através da *Model Update Tool* é possível efectuar o *reverse engineering* de um projecto Visual Basic, criando um novo modelo ou alterando um modelo já existente, neste caso fazendo reflectir no modelo alterações efectuadas ao nível do código.

Por fim, através do *Type Library Importer* é possível importar para o modelo informação sobre componentes COM, que exista numa biblioteca de tipos (*type library*) e que seja referenciada por um projecto Visual Basic. Esta informação é necessária para fornecer uma descrição, independente da linguagem, das interfaces e dos tipos de dados que um componente COM expõe para o exterior.

A cada item de código gerado é atribuído um identificador do modelo (*model ID*). Esta associação possibilita a identificação unívoca de cada elemento do modelo, independentemente de alterações efectuadas ao seu nome, directamente no código. Esta informação não tem qualquer significado para o Visual Basic mas é fundamental para o Rose efectuar o processo de *round-trip engineering* correctamente. Por isso, estes

identificadores não devem em nenhuma circunstância ser alterados. As ferramentas *Code Update Tool* e *Model Update Tool* utilizam estes identificadores para sincronizar o modelo com o código.

Para demonstrar todo este processo vamos utilizar novamente os conceitos do exemplo das compras referido no Capítulo 3, considerando uma versão simplificada do mesmo e utilizando subconjuntos da informação expressa para diferentes casos práticos.

13.7.2 Geração de Código

Definição de Componentes

Uma das primeiras acções a executar é criar na *Component View* um componente, cujo estereótipo seja do tipo programa, e dizer que a linguagem do mesmo será Visual Basic. Esta informação é fundamental para a correcta geração de código, uma vez que mais tarde serão associadas classes a cada componente.

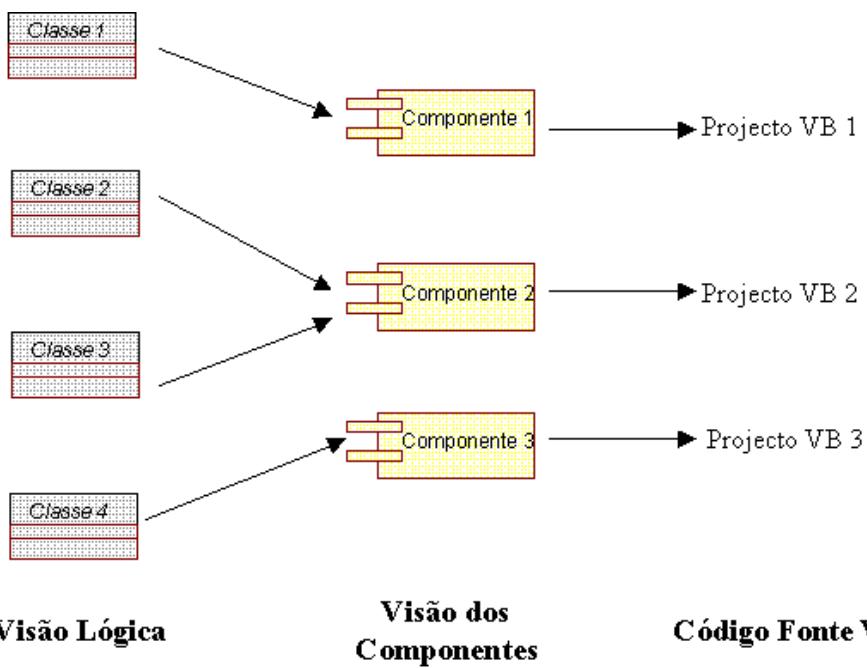



Figura 13.7: Mapeamento entre classes, componentes e projectos Visual Basic.

Dado que o objectivo é gerar um programa executável, a escolha lógica para o estereótipo do componente será “Standard Exe” (outras opções seriam “Activex Exe”, “Activex Dll”, etc). Para além do estereótipo e da linguagem, no tabulador Visual Basic da especificação do componente é apresentado um conjunto de propriedades, das quais a mais significativa é “ProjectFile”, onde se especifica o nome do projecto Visual Basic que estará associado ao componente (Figura 13.8).

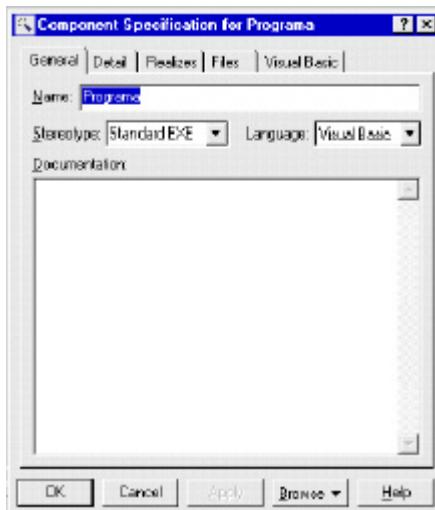



Figura 13.8: Especificação das propriedades de um componente relevantes para a geração de código em Visual Basic.

Definição de Classes

De forma paralela e independente, é importante que as classes sejam definidas e as respectivas especificações sejam elaboradas. No caso do exemplo em análise, vamos considerar apenas as classes `Fornecedores`, `Encomendas` e `Linhas_Encomenda`, segundo o modelo expresso na Figura 13.9. Para além de outras situações, com estas classes podemos demonstrar como são resolvidas as relações de associação do tipo 1 para 1 (entre `Encomendas` e `Fornecedor`) e de 1 para N (entre `Encomendas` e `Linhas_Encomenda`).

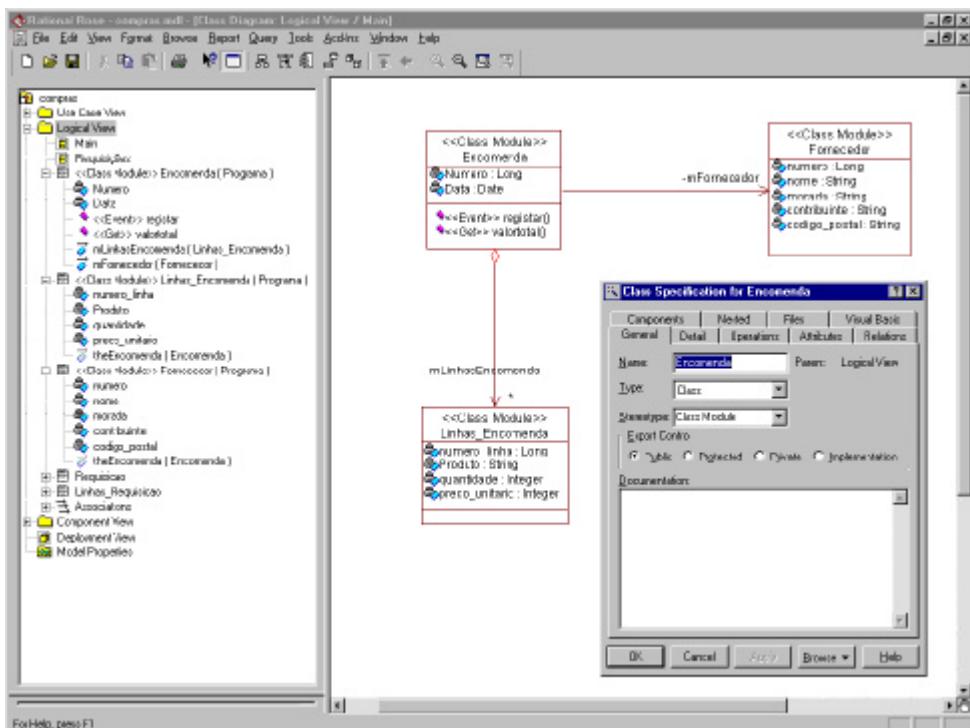



Figura 13.9: Diagrama de classes e especificação da classe Encomenda.

Na definição de uma classe é importante ter em atenção o seguinte:

- A correcta especificação do seu estereótipo, que está relacionado com o tipo de componente a gerar no Visual Basic.
- A correcta especificação de todos os seus atributos.
- A correcta especificação de todos os seus métodos.
- A definição das relações existentes entre as classes.
- A atribuição de valores a algumas propriedades que constam do tabulador Visual Basic e que têm a ver com questões específicas da geração para esta linguagem.

O estereótipo de uma classe funciona como um *template,* que define o tipo de objecto do projecto Visual Basic no qual será mapeado, por exemplo, um “class module” ou um “form”. O *template* inicializa a classe com os membros que são típicos para esse tipo de *project item*. Para gerar código Visual Basic para uma classe do modelo, a classe tem de

estar atribuída a um componente, sendo o Visual Basic a linguagem de implementação desse componente.

O Rose permite ao utilizador definir diversas propriedades de um método, quer na respectiva especificação directa, quer através do *Model Assistant*:

- Um método pode ser classificado como "*Declare*", "*Event*" ou ainda como propriedade ("*Property Get*", "*Property Set*" ou "*Property Let*").
- Os métodos podem ser declarados como *Public (*acessíveis a partir de outros módulos, e concretizado numa função ou subrotina *public); Private (*não acessíveis a partir de outros módulos, e concretizados numa função ou subrotina *private*); *Protected (*acessíveis a partir de outros módulos do próprio projecto, e concretizados numa função ou subrotina *friend)*.
- Os parâmetros do método são declarados com um nome e um tipo de parâmetro. A definição do tipo pode ser omitida.
- Os métodos podem ser estáticos, indicando que as variáveis locais do procedimento são preservadas entre sucessivas invocações.

Uma associação é uma relação bidireccional entre classes, que denota uma dependência semântica entre elas. As associações podem ser descritas em detalhe, através de diversas propriedades disponíveis na respectiva especificação e que são utilizadas pelo gerador de código Visual Basic para determinar o código a gerar. Nomeadamente, é relevante definir a direcção da associação, as funções desempenhadas pelas classes envolvidas, a multiplicidade e navegabilidade da relação.

Por exemplo, na relação entre `Encomenda` e `Fornecedor` é necessário especificar a função da classe `Fornecedor` nesta relação (designada no exemplo por `mFornecedor`), mas não a da classe `Encomenda`. Assim, o código gerado terá uma variável na classe `Encomenda` que representa um `Fornecedor`, enquanto o inverso não acontece.

Geração de Código

Quando esta informação estiver toda correctamente registada, é necessário passar à geração do código propriamente dito, através da ferramenta *Code Update Tool* (Figura 13.10). Durante este processo de

geração, o Visual Basic tem que estar activo, o que é garantido de forma automática pelo Rose.

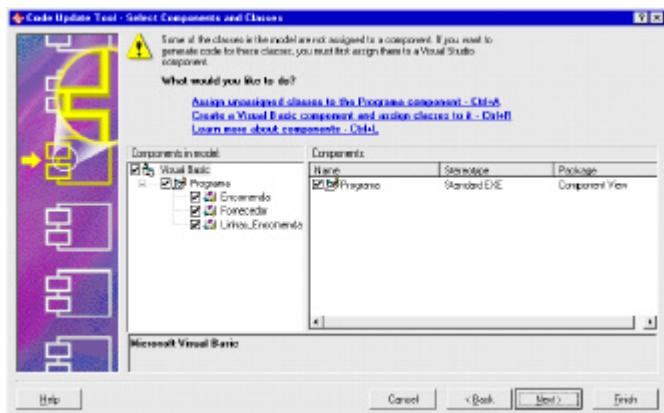



Figura 13.10: Geração de código para Visual Basic.

Durante a geração do código o modelo é ligeiramente alterado, de forma automática, de modo a incluir informação sobre tratamento de erros. Como se pode constatar na Figura 13.11, são acrescentados a cada classe um atributo e uma operação, e é ainda incluído um pacote designado por Debug que contem classes que declaram constantes e definem operações para tratamento de erros.

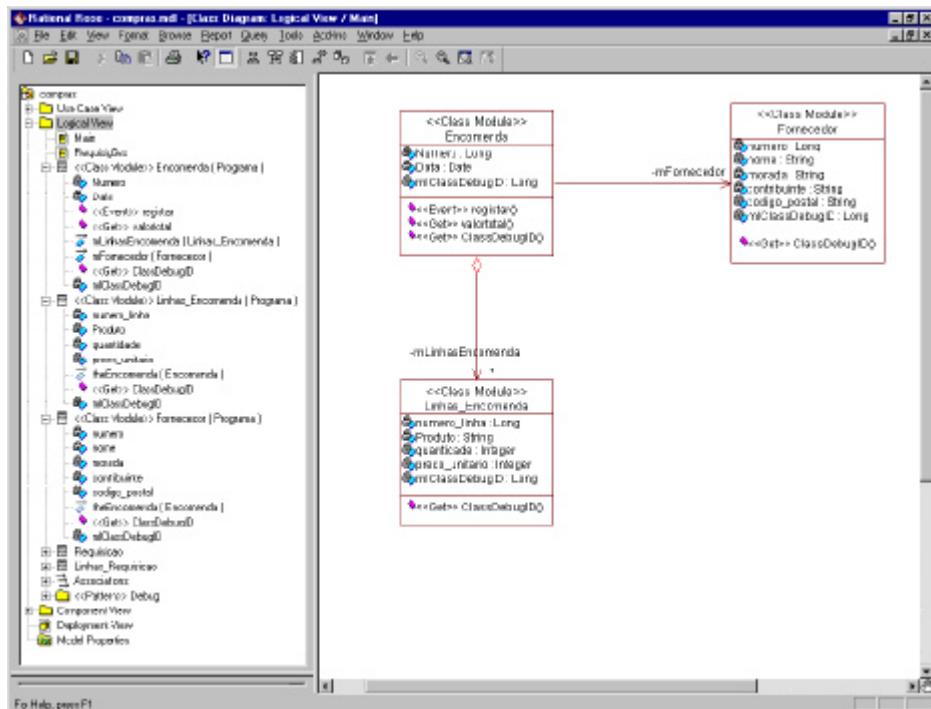



Figura 13.11: Alterações ao modelo depois da geração.

Para cada classe, usando o mapeamento por omissão, o Rose gera a seguinte informação no código:

- Definição de um componente do Visual Basic, a partir do nome e das propriedades da classe no modelo.
- Comentário, produzido a partir da documentação da especificação da classe.
- Variáveis do módulo, que são geradas a partir das relações de associação, agregação e propriedades de classe.
- Especificação de métodos e/ou procedimentos (consoante se esteja a gerar uma classe Visual Basic ou outro tipo de objecto), incluindo o esqueleto do corpo do método e eventual código definido pelo utilizador na especificação de cada método.
- Código de *Debug*, que por omissão é automaticamente gerado para o tratamento de erros dos vários métodos.

No Visual Basic, são automaticamente gerados os ficheiros correspondentes às várias classes do modelo Adicionalmente, são criados alguns componentes que não faziam parte do modelo, e que estão relacionados com tratamento de erros. Enquanto as classes do modelo são mapeadas em classes do Visual Basic, estes novos componentes, devido ao seu estereótipo ser “ClassUtility”, são concretizados por módulos. A Figura 13.12 reproduz o ambiente do Visual Basic para o projecto em análise, após a geração do código, destacando-se o aspecto do código da classe `Encomenda`.

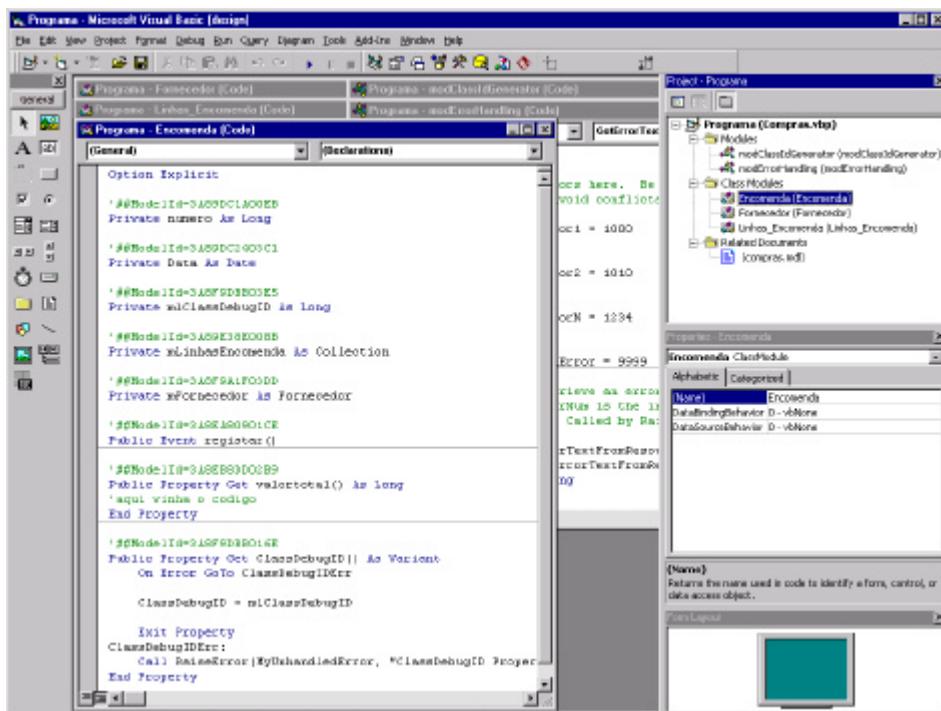
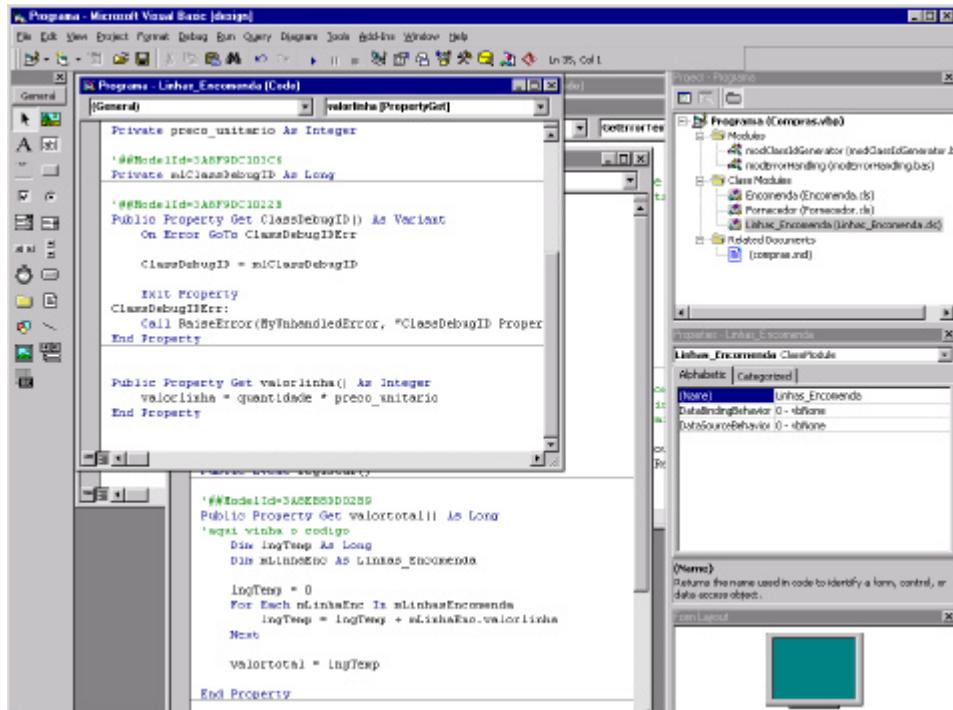



Figura 13.12: Ambiente do Visual Basic após a geração do projecto

13.7.3 Reverse Engineering

O código gerado pelo Rose corresponde, neste caso, integralmente à forma como o modelo deve ser concretizado num projecto Visual Basic. Imaginemos agora que pretendíamos efectuar as seguintes alterações

directamente no código, com o objectivo que fossem posteriormente re-percutidas no modelo:

- Programação da propriedade `valortotal` da classe `Encomenda`, que calcula o custo total da encomenda.
- Criação de uma nova propriedade na classe `Linhas_Encomenda` para calcular o valor total de cada linha da encomenda.

Figura 13.13: Alterações no código das classes Encomenda e Linhas_Encomenda.

O processo de *reverse engineering* no Visual Basic é suportado pela *Model Update Tool,* que actualiza adequadamente o modelo em função das alterações efectuadas ao código. Neste caso, tal significa a criação de uma nova operação na classe `Linhas_Encomenda` (valorlinha) que é classificada correctamente em termos ao seu estereótipo. Para além disso, é importada do Visual Basic toda a informação relativa aos componentes COM que o projecto referenciou quando foi gerado (Figura 13.14).

Para que o código dos diferentes métodos fique sincronizado entre o projecto Visual Basic e o modelo é necessário ter em atenção as propriedades “ReplaceExistingBody” e “DefaultBody”, que constam da especificação de cada operação das classes do modelo. A primeira permite indicar a pretensão de definir código específico para a operação, e a segunda permite escrever esse código. É também nesta propriedade que é colocado o código importado dos componentes Visual Basic, e introduzido directamente nesse ambiente. De programação.

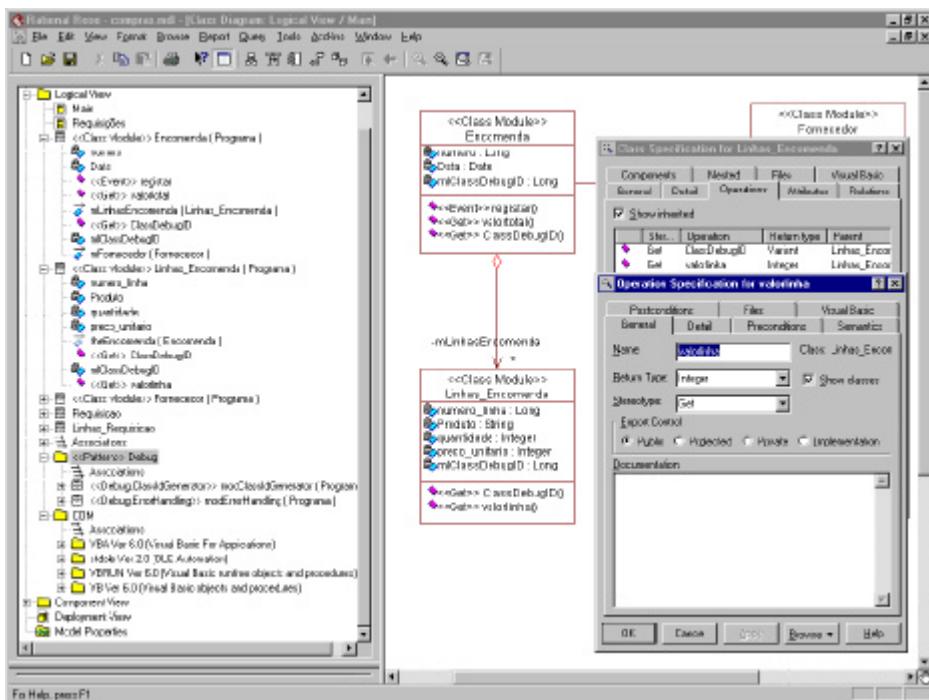



Figura 13.14 : Alterações ao modelo após o processo de Reverse Engineering.

Com esta sequência de passos, em que (1) foi elaborado um diagrama de classes; (2) foi gerado código a partir desse diagrama; (3) foram efectuadas alterações directamente no código e (4) foi importado o código de novo para o modelo, reproduzimos o processo de *round-trip engineering.* Neste exemplo, pudemos constatar que o Rose se comportou adequadamente ao longo de todos os passos.

13.7.4 Relações de Generalização

Uma situação que é interessante analisar tem a ver com o comportamento do Rose na geração de código para uma relação de generalização, e nomeadamente como os atributos e operações públicas, privadas e protegidas são mapeados no Visual Basic.

Figura 13.15: Exemplo de uma relação de generalização.

Uma relação deste tipo é mapeada no Visual Basic através da instrução `implements.` Para além deste facto, é criada na subclasse uma cópia dos métodos públicos da superclasse, e os atributos públicos desta classe são disponibilizados para o exterior através de duas propriedades do Visual Basic, com o mesmo nome, uma "Property Get" e outra "Property Let", em função do especificado no seu estereótipo. Estas alterações são, no decurso do processo da geração de código, reflectidas no modelo (Figura 13.16).

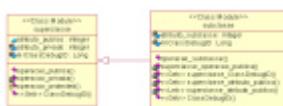



Figura 13.16: Modelo depois do código gerado.

O código das duas classes ficou com o seguinte aspecto (para além do que foi gerado para tratamento de erros):

Superclasse

```
Option Explicit

'##ModelId=3A8FB55B007B
Public atributo_publico As Integer

'##ModelId=3A8FB5680174
Private atributo_privado As Integer

'##ModelId=3A8FB59C0150
Public Sub operacao_publica()
    On Error GoTo operacao_publicaErr

    '## Your code goes here ...

    Exit Sub
operacao_publicaErr:
    Call RaiseError(MyUnhandledError, "operacao_publica Sub")
End Sub

'##ModelId=3A8FB5AB036F
Private Sub operacao_privada()
    On Error GoTo operacao_privadaErr

    '## Your code goes here ...

    Exit Sub
operacao_privadaErr:
    Call RaiseError(MyUnhandledError, "operacao_privada Sub")
End Sub

'##ModelId=3A8FB5B501ED
Friend Sub operacao_protected()
    On Error GoTo operacao_protectedErr

    '## Your code goes here ...

    Exit Sub
operacao_protectedErr:
    Call RaiseError(MyUnhandledError, "operacao_protected Sub")
End Sub
```

Subclasse

```
Option Explicit

'##ModelId=3A8FB2C901C9
Implements superclasse

'##ModelId=3A8FB64C03CA
Private atributo_subclasse As Integer

'##ModelId=3A8FB622003B
Public Sub operacao_subclasse()
    On Error GoTo operacao_subclasseErr
    '## Your code goes here ...

    Exit Sub
operacao_subclasseErr:
    Call RaiseError(MyUnhandledError, "operacao_subclasse Sub")
End Sub

'##ModelId=3A8FBE03026C
Private Sub superclasse_operacao_publica()
    On Error GoTo superclasse_operacao_publicaErr
    '## Your code goes here ...

    Exit Sub
superclasse_operacao_publicaErr:
    Call RaiseError(MyUnhandledError,
"superclasse_operacao_publica Sub")
End Sub

'##ModelId=3A8FBE04010F
Private Property Get superclasse_atributo_publico() As Integer
    On Error GoTo superclasse_atributo_publicoErr
    '## Your code goes here ...

    Exit Property
superclasse_atributo_publicoErr:
    Call RaiseError(MyUnhandledError,
"superclasse_atributo_publico Property")
End Property

'##ModelId=3A8FBE04028B
Private Property Let superclasse_atributo_publico(ByVal RHS As
Integer)
    On Error GoTo superclasse_atributo_publicoErr
'## Your code goes here ...

Exit Property
superclasse_atributo_publicoErr:
```

```
            Call RaiseError(MyUnhandledError,
"superclasse_atributo_publico Property")
        End Property
```

Podemos observar que tudo o que é herdado da superclasse fica na subclasse com visibilidade privada, o que não deveria acontecer. Adicionalmente, o método para tratar erros da superclasse e também "herdado" pela subclasse, o que não faz qualquer sentido. Podemos pois concluir que a geração de código numa relação de generalização não é efectuada de forma adequada.

13.7.5 Comentários à Geração de Código

As funcionalidades de geração de código e de *reverse engineering* do Rose são, de um modo geral, bastantes completas e robustas. As regras de mapeamento de Visual Basic para Rose para suportar o processo de *reverse engineering* encontram-se também bem definidas, e podem ser consultadas detalhadamente nos manuais de referência do produto. O conceito de estereótipo assume em todo o processo uma importância muito relevante, pois é com base nele que é efectuado o mapeamento entre os artefactos do Rose e os componentes do Visual Basic.

No entanto, existem alguns problemas que não são adequadamente solucionados. Um tem a ver com a inexistência de suporte para a geração da interface gráfica, o que é particularmente importante no caso de ferramentas de modelação que geram código para ambientes de desenvolvimento gráficos, como é o caso do Visual Basic. No caso desta ferramenta, o Rose permite gerar o código para um componente do tipo "Form", a partir de uma classe com o estereótipo "Form", no entanto toda a parte visual não pode ser gerada.

Um outro aspecto tem a ver com algumas deficiências no processo de geração e *reverse engineering*, relacionadas com o correcto mapeamento de relações do modelo no código. É o caso das relações de generalização, cuja concretização em Visual Basic implica a geração de código com certas inconsistências. No entanto, é nossa opinião que tal

acontece porque o Visual Basic não é uma linguagem verdadeiramente orientada por objectos, tal como a definimos na Secção 3.5, mas apenas baseada em componentes. Neste sentido, não suporta adequadamente certas funcionalidades como sejam a definição de herança e o polimorfismo. Este tipo de problemas não acontece, por exemplo, em Java ou C++.

13.8 Geração de Modelos de Dados

O Rose disponibiliza várias funcionalidades para a geração de esquemas de bases de dados, essencialmente divididas num processo de geração genérico para várias bases de dados que suportem o SQL standard, normalizado pela organização ANSI, e num outro, mais completo, mas especifico para Oracle versão 8.

No primeiro caso as funcionalidades de modelação estão relacionadas com o trabalho realizado pelo comité ANSI/SPARC, que, numa tentativa de normalizar a arquitectura dos modelos de dados das bases de dados, propôs que estes fossem divididos em três níveis:

- O nível externo é o que está mais próximo do utilizador, e preocupa-se em especificar a percepção que os utilizadores têm dos dados, através das aplicações e das interacções que efectuam com os dados.
- O nível interno é o que está mais próximo do dispositivo físico de armazenamento, sendo, por vezes, referido como nível físico.
- O nível conceptual é uma forma de estabelecer uma correspondência entre os outros dois níveis, e é concretizado através da *Data Definition Language* (DDL).

A versão actual do UML não contempla quaisquer mecanismos de base para a produção de informação para os níveis físico e conceptual. No nível externo, esta linguagem é bastante completa: são utilizados os diagramas de classes e de colaboração, que permitem identificar os conceitos e as entidades da organização e respectivas relações.

Percebendo a importância que a modelação de dados, ao nível conceptual, assume nas ferramentas CASE, a Rational disponibiliza até à versão Rose 2000 um *add-in* através do qual é possível gerar

esquemas DDL de criação da base de dados para Oracle, SQL Server, Sybase, Watcom e ANSI SQL. Suporta também funcionalidades mais específicas no caso da base de dados Oracle, através de um gerador específico. Adicionalmente, existe a possibilidade de desenvolver uma interface com outras ferramentas mais especializadas nesta área, como é o caso do ERwin (*www.cai.com*).

No entanto, e como esta opção não produz resultados completamente satisfatórios, devido à diferença existente entre os conceitos de modelação UML e os relacionados com bases de dados, a Rational optou recentemente por propor na versão Rose 2001 (que saiu durante a conclusão deste livro e que portanto não foi possível avaliar) o *UML Profile for Data Modeling* [Rational00]. Um perfil UML, tal como foi definido na Secção 9.4, é um método aprovado pela OMG para adicionar funcionalidades ao UML numa área específica, sem necessidade de alterar o metamodelo UML. Este perfil não se encontra reconhecido pela OMG.

Até à versão 2000, o Rose não suportava a funcionalidade de *reverse engeneering* a partir de um esquema da base de dados, com a excepção de funcionalidades específicas para Oracle.

13.8.1 Geração de Modelos de Dados até ao Rose 2000

Tal como na geração de código, também os modelos de dados das bases de dados são gerados a partir do conceito de classe, que é concretizado no conceito de tabela, e dos respectivos atributos que constituem as colunas das tabelas. As relações entre as classes permitem definir chaves externas, sendo esta uma das áreas em que o Rose mais lacunas apresentava.

No caso específico do Oracle 8 são ainda suportadas algumas funcionalidades adicionais, nomeadamente: (1) a possibilidade de importar os tipos de dados suportados no SGBD Oracle (a partir do *framework* referido na Secção 13.1); (2) a verificação do código de um esquema e posterior geração; e (3) importação da estrutura de uma base de dados existente.

Os passos necessários para a geração de um esquema de uma base de dados são os seguintes:

- Definição das classes, respectivos atributos e relações. As classes para as quais se pretender gerar uma tabela deverão ser definidas como persistentes.
- Definição de um conjunto de propriedades gerais, que afectam a geração do referido esquema e que são designadas por propriedades do projecto. Estas características deverão ser alteradas na opção *Tools-Options*, no tabulador intitulado DDL, e seleccionando na *combobox Type* a opção "Project" (ver Figura 13.17).
- Definição de propriedades de diversos tipos de dados, que interessam para mapear os atributos das classes em colunas de tabelas (por exemplo, chaves primárias). Esta parametrização prévia deve ser efectuada na mesma opção indicada no ponto anterior, seleccionando na *combobox Type* a opção *Attribute*.
- Selecção das classes para as quais se pretende gerar o código de criação da estrutura das bases de dados.
- Geração do código do esquema de criação de tabelas, para uma base de dados a indicar.

Definição de Propriedades Gerais

Figura 13.17: Especificação de parâmetros para a geração do script de uma base de dados em Sql Server.

As propriedades do projecto cuja especificação é relevante incluem, entre outras (Figura 13.17):

- *DataBase*: indica o tipo de sistema de gestão de bases de dados por omissão, para o qual o esquema será gerado. Os valores possíveis são Oracle, SQL Server, Sybase, Watcom e ANSI SQL.

- *PrimaryKeyColumnName*: é a extensão concatenada ao nome da classe para a criação do nome da chave primária, no caso em que a chave primária é atribuída pelo Rose.
- *PrimaryKeyColumnType*: é o tipo de dados utilizado para a chave primária gerada pelo Rose.
- *ViewName*: possibilita a definição (opcional) do prefixo a concatenar a todas as vistas geradas: Se não for definido, não é concatenado nenhum prefixo.
- *TableName*: possibilita a definição (opcional) do prefixo a concatenar a todas as tabelas geradas. Se não for definido, não é concatenado nenhum prefixo.
- *InheritSuffix*: é um sufixo adicional (opcional) a ser concatenado aos nomes das vistas que são produzidas para mapeamento de herança.
- *DropClause*: é um valor lógico que, se for verdadeiro, cria um comando DROP TABLE antes de cada comando CREATE TABLE.
- *DDLScriptFileName*: é o nome atribuído, por omissão, ao ficheiro a gerar e que contem o *script* DDL.

Definição de Tipos de Dados

O passo da especificação de tipos de dados a utilizar e a associar a cada atributo é particularmente importante, pois a geração dos esquemas DDL não tem em conta o tipo de dados dos atributos da classe, mas aquele que estiver especificado na informação que consta do tabulador DDL. Por omissão, é utilizado um conjunto de valores que gera o tipo de dados VARCHAR (apesar de se seleccionar a base a dados para a qual o esquema é gerado, o Rose não possui, por omissão, qualquer informação de validação relativamente aos tipos de dados suportados por cada SGBD).

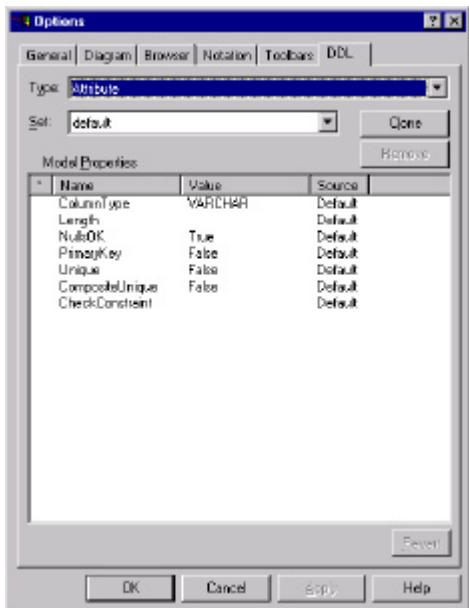
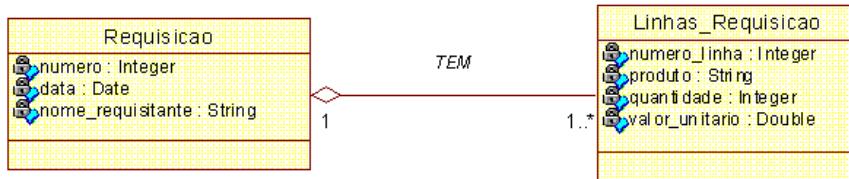



Figura 13.18: Conjunto de atributos que podem ser aplicados a um tipo de dados

A definição de um conjunto de *atributes sets* correcto permite tirar maior partido da geração automática e com menos alterações manuais; estes atributos incluem (ver Figura 13.18):

- *ColumnType*: permite indicar o tipo de dados da coluna.
- *Length*: indica o tamanho do tipo de dados da coluna, se aplicável.
- *NullsOk*: indica se a coluna permite valores nulos.
- *PrimaryKey*: indica se a coluna é chave primária.
- *Unique*: indica se os valores da coluna deverão ser únicos.

Exemplo 1: Geração de uma Classe

De forma a exemplificar a geração de código de criação de uma base de dados, analisemos como o Rose trata um conjunto de situações relativamente simples, com base no diagrama de classes da Figura 13.19. No primeiro caso vamos considerar apenas uma tabela, mantendo a associação do *atribute set* atribuído por omissão a todos os atribu-

tos da classe, e corrigindo de seguida esta situação. O SGBD escolhido para geração foi o Sql Server.

Figura 13.19: Exemplo de relação entre as classes Requisição e Linhas_Requisção.

Se considerarmos apenas a classe `Requisicao` e gerarmos o código de criação da tabela correspondente após termos definido os seus atributos e indicando os tipos de dados, obtém-se o esquema da tabela que consta no exemplo seguinte.

```
DROP TABLE Requisicao
go

CREATE TABLE T_Requisicao(
    numero VARCHAR,
    data VARCHAR,
    total VARCHAR,
    nome_requisitante VARCHAR,
    RequisicaoId NUMBER(8),
    PRIMARY KEY(RequisicaoId))
go
```

Como se constata, os tipos de dados das colunas não correspondem aos homólogos definidos na classe. Isto acontece porque é preciso alterar os *atribute sets* para cada atributo da classe, o que é efectuado na respectiva especificação, no tabulador DDL (Figura 13.20). Além disso, e como não tinha sido definido nenhum atributo do modelo como chave primária, o Rose encarregou-se de adicionar um novo atributo à tabela T_Requisicao, que funciona como chave primária.

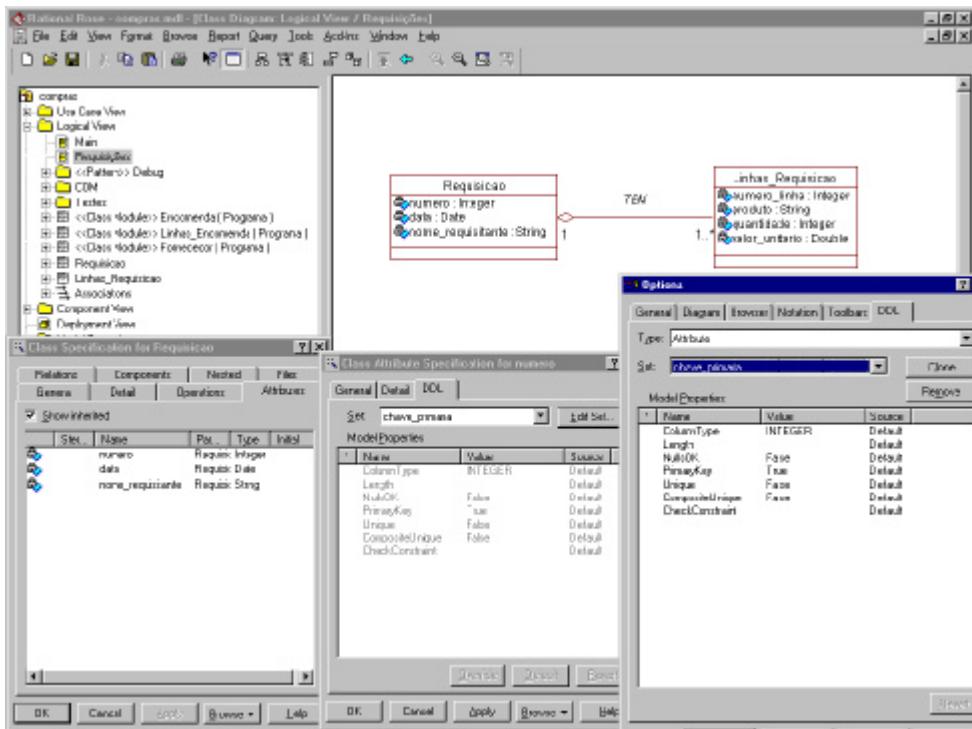



Figura 13.20: Alteração das propriedades dos atributos para a criação da tabela.

Na Figura 13.20, existe uma sequência na ordem dos ecrãs, da esquerda para a direita: o primeiro ecrã é o da especificação dos atributos da classe `Requisicao`; o segundo é o da especificação do atributo `numero`, que pretendemos que seja a chave primária; e o terceiro é o ecrã de definição das propriedades comuns a um determinado conjunto de atributos, que neste caso definimos como sendo "todos os atributos que são chaves primárias".

Com a alteração dos *attribute sets* para cada atributo, a geração do esquema DDL para a tabela `Requisicao` produz os seguintes resultados:

```
DROP TABLE T_Requisicao
go
CREATE TABLE T_Requisicao(
    numero INTEGER NOT NULL,
    data DATETIME,
    total INTEGER,
    nome_requisitante VARCHAR,
    PRIMARY KEY(numero))
go
```

Exemplo 2: Geração de Classes Relacionadas

Se a geração de tabelas isoladas obedece às regras habituais do modelo relacional, já o mesmo não se pode dizer a partir do momento em que se definem relações entre classes, particularmente as relações de associação e de generalização. Numa relação de associação, são geradas duas tabelas, uma para cada classe. Em cada tabela são colocadas referências para a chave primária da outra tabela, como se ilustra de seguida.

```
DROP TABLE T_Linhas_Requisicao
go
CREATE TABLE T_Linhas_Requisicao(
    numero_linha VARCHAR,
    codigo_produto INTEGER,
    quantidade INTEGER,
    valor_unitario INTEGER,
    valor_total INTEGER,
    numero INTEGER NOT NULL,
    FOREIGN KEY (numero) REFERENCES T_Requisicao,
    Linhas_RequisicaoId INT,
    PRIMARY KEY(Linhas_RequisicaoId))
go

DROP TABLE T_Requisicao
go
CREATE TABLE T_Requisicao(
    numero INTEGER NOT NULL,
    data DATETIME,
    total INTEGER,
```

```
        nome_requisitante VARCHAR,
        TEM INT REFERENCES T_Linhas_Requisicao(Linhas_RequisicaoId),
        PRIMARY KEY(numero))
    go
```

Para além dos problemas de coerência a nível do modelo relacional, existem também outras situações em que a geração não se comporta adequadamente, tais como:

- Na geração de código para suportar relações de generalização. Neste caso, é criada uma vista que relaciona as duas tabelas, quando na realidade o que seria de esperar era a inclusão da chave primária da tabela da superclasse na tabela da subclasse.
- Quando se pretende utilizar a mesma classe para gerar código para uma linguagem de programação, e para a criação do esquema DDL correspondente. Isto porque não é possível especificar simultaneamente propriedades de geração para as duas situações, uma vez que uma classe só pode estar associada a um componente.

13.8.2 Geração de Dados a partir do Rose 2001

Precisamente para solucionar os problemas e lacunas identificados, a partir da versão 2001 o Rose inclui um componente designado *Data Modeler*, no sentido de facilitar a modelação específica de dados. O *UML Data Modeling Profile* suportado pelo *Data Modeler* baseia-se na definição de estereótipos adicionados às estruturas UML existentes:

- Estereótipos acrescentados às estruturas do UML como sejam os componentes, os pacotes e as classes que permitem a modelação de bases de dados, esquemas e tabelas.
- Estereótipos acrescentados às associações do UML que permitem a modelação de relações.
- Estereótipos acrescentados às operações do UML que permitem a modelação de chaves primárias, chaves externas, restrições de valores únicos, *triggers* e *stored procedures*.

É ainda adicionado um novo tipo de diagrama, o **diagrama do modelo de dados**, que é uma forma lógica de apresentar tabelas, colunas e relações criadas a partir da informação que consta do modelo de objectos. Este diagrama é gerado a partir de um diagrama de classes, sendo incluídas todas as classes definidas como persistentes. Para cada diagrama do modelo de dados, é possível especificar as propriedades das tabelas e das colunas, e uma vez que o mapeamento entre modelos de objectos e de dados não é completo, é necessário, por vezes, efectuar algumas alterações directamente no modelo de dados, de forma a normalizá-lo. A partir deste diagrama pode-se gerar um esquema da base de dados. Podem ainda ser adicionadas outras propriedades, como restrições (*constraints*) e índices, que são conceitos específicos do mundo das bases de dados.

Esta última versão do Rose permite efectuar *reverse engineering* da estrutura de uma base de dados, e reflecti-la num diagrama de dados.

Ao nível da estrutura, o mapeamento entre o modelo aplicacional e o modelo de dados é relativamente intuitivo:

- A **base de dados** encontra-se associada a um **componente** na *component view* do Rose (de forma consistente com o que acontece actualmente com um projecto Visual Basic, também associado a um componente).
- Um **pacote** é mapeado num **esquema**, e é este mapeamento que suporta directamente todo o processo de *forward* e *reverse engineering* da base de dados.
- As **classes** são mapeadas em **tabelas**, normalmente numa relação de uma tabela para cada classe, mas que pode ser diferente no caso de classes com associações.
- Os **atributos** são mapeados em **colunas** de uma tabela, sendo importante a conversão dos tipos de dados entre o nível aplicacional e os utilizados nas bases de dados, que variam ligeiramente entre diferentes servidores de bases de dados.

O mapeamento das relações de associação para as bases de dados pode ser mais complexo, consoante o tipo de cardinalidade da relação. Assim:

- Associações de 1 para 1 entre duas classes dão origem a duas tabelas em que a chave primária de uma é também chave externa da outra.
- Associações de 1 para N entre duas classes dão origem a duas tabelas em que a chave primária da tabela principal é colocada como chave externa na outra.
- Associações de M para N entre duas classes dão origem a três tabelas em que as chaves primárias das tabelas que mapeiam directamente as classes são colocadas numa terceira tabela, tal como aconteceria num processo de normalização do modelo relacional, garantindo assim a integridade do modelo.

A generalização entre duas classes (superclasse e subclasse) é mapeada em duas tabelas, em que a chave primária da tabela correspondente à superclasse é também chave primária e externa da outra tabela.

Estes factos permitem-nos concluir que, pelo menos do ponto de vista teórico e dos conceitos, a última versão do Rose veio corrigir os problemas existentes na versão Rose 2000.

13.9 Geração da Interface Homem-Máquina

O UML não tem funcionalidades específicas para a geração da interface de programas para os utilizadores, quer esta seja gráfica quer orientada ao caracter. No caso específico do Visual Basic, dispõe da possibilidade de caracterizar o estereótipo de uma classe como sendo do tipo "Form", verificando-se que de facto é gerado correctamente um componente Form do Visual Basic, com os métodos que tiverem sido definidos. Podem ainda ser definidos os controles que ficarão associados ao Form, ao nível do modelo, também com estereótipos próprios. Contudo, a parte gráfica, de colocação dos controles no Form terá que ser efectuada no Visual Basic.

13.10 Geração de Documentação

A documentação é uma parte importante do processo de desenvolvimento de software. O Rose oferece diversas possibilidades de elaboração de documentação, nomeadamente: (1) a utilização de relatórios pré-definidos; (2) a criação de novos relatórios através de *scripts*; (3) a utilização de ferramentas especificas para a produção de relatórios; e (4) a geração de informação para publicação na Internet.

Os relatórios pré-definidos, em número não muito elevado, apresentam o seu *output* em formato Word, e incluem informação sobre o esquema dos modelos, classes, comentários, operações, atributos, etc. Existem também vários relatórios específicos para a base de dados Oracle. Para completar este conjunto, o Rose disponibiliza várias ferramentas adicionais que permitem completar as suas capacidades, designadamente:

- Rose Web Publisher: para geração de documentação em formato HTML.
- SoDA: ferramenta da Rational para produção de documentação.
- *Scripts* de geração de relatórios, com a possibilidade de geração de relatórios para outros formatos, como por exemplo Excel.

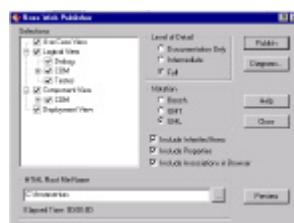
Se bem que a documentação base seja muito simples, os relatórios produzidos são relativamente completos em termos do conteúdo dos diversos modelos. A geração é efectuada a partir da informação introduzida na janela apresentada na Figura 13.21.

Figura 13.21: Geração de documentação do Rose.

13.10.1 Ferramenta SoDA

O Rose pode interagir de forma controlada com ferramentas de documentação externas. O sistema proposto pela Rational é designado por SoDA (*www.rational.com/products/soda*), o qual automatiza a produção da documentação do software, reduzindo substancialmente o esforço necessário para a sua elaboração. Segundo a Rational, o SoDA pode ser integrado com qualquer outra ferramenta, de modo a satisfazer todas as necessidades de documentação de um projecto.

Para além de características como a flexibilidade e a automatização, a ferramenta SoDA apresenta ainda as seguintes funcionalidades:

- Personalização de *templates* de documentos, usando características *WYSIWYG* e caixas de diálogo gráficas.
- Geração de documentos a partir de outras ferramentas de engenharia de software, de forma automática, a partir de regras configuráveis.
- Edição de documentos, recorrendo a funcionalidades poderosas de publicação, de manipulação de texto, inclusão de gráficos e tabelas adicionais, etc.
- Revisão de documentos.
- Publicação de documentos estabilizados.
- Integração com processadores de texto, nomeadamente com o Word.
- Preservação, entre versões sucessivas de um dados documento, de informação introduzida de forma manual, directamente no documento.

- Extracção de dados de várias fontes.
- Suporte a diversos standards, nomeadamente, ISO 9000.

Uma das características diferenciadoras do SoDA é a capacidade de efectuar alterações, associada à possibilidade de desenvolver documentos de forma iterativa, em simultâneo com o desenvolvimento do próprio sistema. Para além das características já enumeradas, o SoDA apresenta os seguintes benefícios:

- Reduz o trabalho manual de edição necessário para criar e manter documentação.
- Reduz o tempo de produção da documentação.
- Aplica uma abordagem iterativa à elaboração de documentação.
- Mantem a consistência entre os documentos e a informação de modelação.
- Facilita a manutenção de documentação.

13.10.2 Rose Web Publisher

O Rose Web Publisher permite criar uma versão HTML de um modelo do Rose. Deste modo, é possível visualizar toda a informação de um modelo, utilizando um browser Internet. Igualmente, é possível publicar iterações sucessivas de um modelo, de modo a rever ou a partilhar a informação. O Rose Web Publisher reproduz os vários elementos presentes no Modelo, incluindo diagramas, classes, pacotes, relações, atributos e operações.

Figura 13.22: Rose Web Publisher.

O Web Publisher pode ser configurado de modo a publicar apenas um subconjunto da informação do modelo. Por exemplo, pode-se selec-

cionar o conjunto de visões que se pretende publicar, o nível de detalhe a incluir, a notação a utilizar (ver Figura 13.22).

13.10.3 Scripts de geração de relatórios

Através de programação, é possível melhorar algumas das capacidades de geração de documentação do Rose. De facto, já os relatórios base disponíveis não são mais do que um conjunto de *scripts*, desenvolvidos em RoseScript.

O exemplo seguinte apresenta um *script* RoseScript, que efectua a geração automática de documentação de todas as classes de um determinado modelo.

```
'---------------------------------------------------------------
'ExportDoc.ebs
'
'      This script exports documentation from a rose model to a
 text file.  The file is
'      created with the following format:
'
'      "Classname", "Class documentation"
'      "Next class name", "Class documentation"...
'
'---------------------------------------------------------------
Sub ExportClassDocumentation (FileName As String)
       Dim AllClasses As ClassCollection
       Dim theClass As Class

       Set AllClasses = RoseApp.CurrentModel.GetAllClasses ()

       Viewport.Open
       Open FileName$ For Output Access Write As #1
       For ClsID = 1 To AllClasses.Count
              Set theClass = AllClasses.GetAt (ClsID)
              Write #1, theClass.Name, theClass.Documentation
       Next ClsID
       Close #1
       Print "Done!"
End Sub

Sub Main
```

```
            FileName$ = SaveFileName$ ("Export Class Documentation",
"Text files:*.txt")

            If FileName$ <> "" Then    ExportClassDocumentation
FileName$
      End Sub
```

13.11 Conclusão

O Rose é sem dúvida uma ferramenta a considerar por quem pretende utilizar a linguagem UML para a modelação de software. É uma ferramenta completa e robusta, em termos da aplicação das regras e princípios do UML, bem como do desenho dos diversos diagramas. A sua aprendizagem básica é intuitiva e rápida. Outro ponto forte evidenciado pelo Rose é a sua capacidade de extensibilidade, dificilmente realizável com outras ferramentas.

Por outro lado, as suas maiores lacunas situam-se em áreas para as quais o próprio UML não dispõe de suporte, nomeadamente: (1) a modelação de processos de negócio e (2) a modelação de bases de dados. Esta última pode ter sido ultrapassada na versão recentemente anunciada e disponibilizada, o Rose 2001, com a inclusão do *UML Profile for Data Modeling*. Quanto à primeira lacuna, é natural que algumas das extensões venham gradualmente a ser implementadas directamente no Rose, e que esta ferramenta estabeleça interfaces para acesso a outras ferramentas de modelação do negócio para a realização de funções especificas como a simulação.

De qualquer modo, o Rose beneficiará sempre da vantagem competitiva que representa o facto de ser comercializado pela empresa que "inventou" o UML.

Capítulo 14 - SYSTEM ARCHITECT

Tópicos

- Introdução
- Interface Gráfica
- Repositório
- Técnicas de Modelação
- Modelação do Negócio
- Geração de Código – Caso de Estudo em Java
- Geração de Modelos de Dados
- Geração de Interfaces Homem-Máquina
- Mecanismos de Extensibilidade
- Geração de Documentação
- Conclusão

14.1 Introdução

O nosso objectivo com este capítulo sobre o System Architect é apresentar uma ferramenta que, desde há vários anos, tem desempe-nhado um papel importante na aplicação de técnicas de modelação, de âmbito muito abrangente (porque disponibiliza técnicas para modelação do negócio, modelação de processos, modelação de dados), e que incorporou com sucesso as notações do UML. Nesse sentido, procura-mos demonstrar as características do System Architect, relativamente

aos mesmos critérios analisados no capítulo anterior sobre o Rose, designadamente:

- Modelação de processos de negócio.
- Geração automática de documentação.
- Geração de código, *reverse engineering* e *round-trip engineering*.
- Mecanismos de extensibilidade disponibilizados.

O System Architect é uma ferramenta desenvolvida pela empresa Popkin Software. Apareceu inicialmente no mercado como ferramenta de modelação de software, recorrendo a técnicas estruturadas, quer de modelação de processos quer de dados. Ao longo do tempo, o System Architect foi evoluindo e incorporando outras notações (modelação de negócio e modelação orientada por objectos). As suas últimas versões passaram a incluir também a linguagem UML, sem nunca deixar de suportar as anteriores notações. É natural que a posição de destaque que tinha assumido no mercado das ferramentas de modelação que utilizavam abordagens estruturadas, lhe permita desempenhar um papel importante nas ferramentas de modelação em UML.

Ao contrário do Rose, que aparece integrado num conjunto de ferramentas mais vasto, sendo inclusivamente a ferramenta de suporte às actividades de modelação de uma metodologia (o RUP), o System Architect é sobretudo uma ferramenta de modelação, que, por isso, pretende ser o mais abrangente possível, suportando um conjunto de técnicas muito vasto.

Uma vez que o System Architect suporta diversas técnicas de modelação, para além do UML, faremos uma breve referência a elas, até porque o System Architect recorre a essas técnicas nalgumas situações para resolver determinados problemas não solucionáveis em UML.

O System Architect tem disponíveis versões vários sistemas operativos, designadamente para ambientes Windows e Unix. Para a elaboração deste capítulo utilizou-se a versão de avaliação System Architect 2001 para Windows, disponível no *Website* do fornecedor (*www.popkin.com*) para *download*.

Ao nível das técnicas de modelação, as principais funcionalidades disponibilizadas pela ferramenta podem ser agrupadas do seguinte modo:

- Modelação de processos de negócio.
- Modelação de software segundo abordagens estruturadas.
- Modelação de dados, com possibilidade de geração de esquemas de bases de dados e *reverse engineering*.
- Modelação de software segundo abordagens orientadas por objectos, particularmente UML.
- Geração de código configurável, com suporte para as seguintes linguagens e/ou ambientes de desenvolvimento: C++, Visual Basic, Powerbuilder, Delphi, Smalltalk, Corba IDL, Java, JavaScript e HTML.
- *Reverse engineering* nativo de C++, Smalltalk, Visual Basic e Java, com possibilidades de incluir outras linguagens.
- Integração entre técnicas de modelação do negócio, UML e modelação de dados.
- Geração de interfaces gráficas e orientadas ao caracter.
- Geração configurável de relatórios, com particular destaque para a documentação em Microsoft Word e para HTML.
- Repositório configurável e extensível.
- Suporte ao trabalho em equipas.

A Popkin tem ultimamente desenvolvido esforços significativos no sentido de melhorar as funcionalidades disponíveis para a modelação de processos de negócio, quer através da representação de diversos tipos de diagramas (nomeadamente, diagramas de processos de negócio, da estrutura da organização, etc.) quer pela integração de técnicas de análise de custos de actividades segundo o modelo ABC (*Activity Based Costing*), e construção de diversas matrizes para analisar as dependências existentes entre conceitos de modelação.

De forma a facilitar o suporte para a modelação de processos de negócio electrónico, a Popkin tem vindo a desenvolver funcionalidades que incluem o suporte ao XML e a novas ferramentas de modelação e/ou desenvolvimento para a Internet.

Como a maioria das ferramentas de modelação de software, que suportam múltiplas notações e técnicas de modelação, o System Architect não privilegia a utilização de nenhuma metodologia. Esta flexibilidade permite que, num mesmo projecto, seja possível utilizar técnicas de diferentes abordagens, em função da parametrização efectuada quando se cria um novo projecto. Por exemplo, pode-se utilizar um diagrama de fluxo de dados do SSADM para modelar os processos da organização e um diagrama de classes em UML para representar os conceitos do negócio.

14.2 Interface Gráfica

Depois de criado um novo projecto, o aspecto da interface gráfica do System Architect é o apresentado na Figura 14.1.

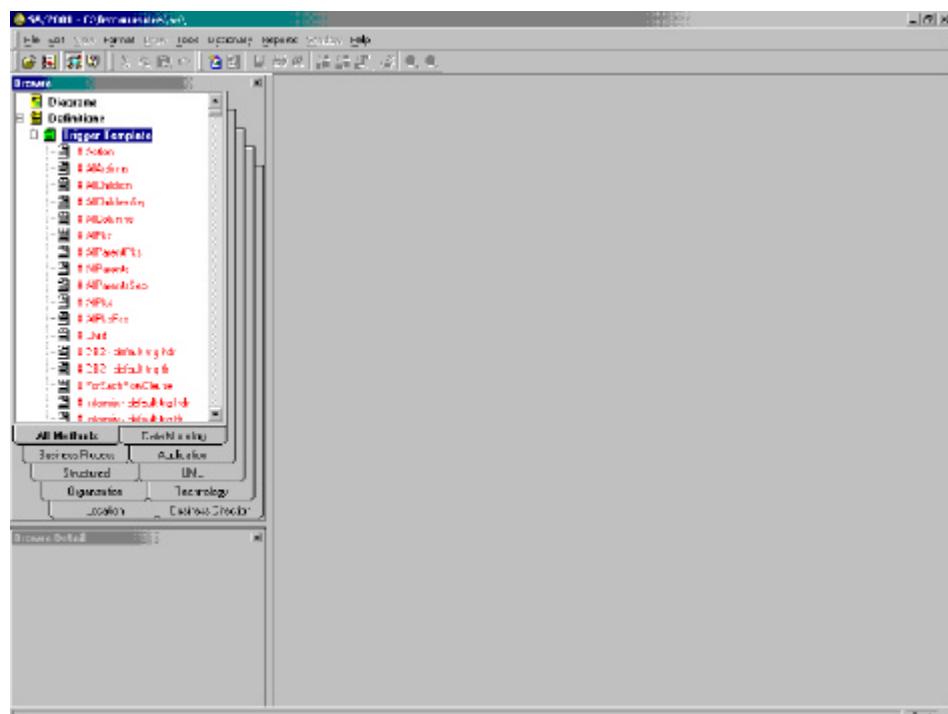



Figura 14.1: Interface de um novo projecto.

Para além dos tradicionais menus de opções e *toolbar* com as funcionalidades mais comuns, a interface do System Architect encontra-se dividida em duas grandes áreas:

- O *browser*, que disponibiliza as funcionalidades de navegação nos elementos do repositório, e que se encontram agrupados em diagramas e definições.
- A área de desenho dos diversos diagramas, que permite a coexistência de vários diagramas simultaneamente abertos.

Adicionalmente, do lado inferior esquerdo existe uma janela designada por *browse detail*, que apresenta informação mais detalhada sobre o elemento seleccionado no *browser*: no caso deste se tratar de um diagrama, ele é apresentado em formato reduzido; no caso de ser uma definição, apresenta uma lista de propriedades e os valores em vigor para cada uma delas.

Se ao nível da interface e das respectivas áreas existe uma semelhança entre o Rose e o System Architect, podemos também constatar desde já algumas diferenças importantes. Nomeadamente, o Rose tem ao nível do *browser* a preocupação de categorizar os artefactos em função das diferentes visões que um interveniente no processo de desenvolvimento pode ter; o System Architect providencia a navegação nos artefactos com base numa classificação em tipos de diagramas e de definições, portanto de forma mais pragmática.

Para além disso, como o System Architect tem um número relativamente elevado de diagramas e de definições, providencia, adicionalmente, a sua categorização em função do tipo de modelos / notações aos quais qual o diagrama (ou a definição) diz respeito. É essa a função dos vários tabuladores (*Data Modeling*, *Business Process*, *Application*, etc).

O elevado número de métodos providenciados pelo System Architect, pertencentes a diferentes abordagens e áreas de modelação, pode significar uma curva de aprendizagem longa, pois nem sempre é fácil ao utilizador decidir quais os métodos a utilizar, e o impacto do preenchimento de determinado parâmetro de configuração. Esta estratégia de elevada abrangência de técnicas de modelação faz com que nem

sempre tenha sido possível ao fabricante acompanhar o ritmo de evolução da tecnologia. Só assim se justifica que, actualmente, algumas das funcionalidades de geração de código disponíveis para Visual Basic são, ainda, para a versão 4 desse produto (apesar de também já ter uma opção de *round-trip engineering* para Visual Basic 6)!

14.3 Repositório

A criação de um novo projecto passa, em primeiro lugar, pela indicação da localização física do repositório do System Architect (na terminologia da ferramenta é designado por "enciclopédia"), que não é mais do que uma directoria onde se encontram gravados os diversos ficheiros envolvidos (Figura 14.2). Cada directoria é utilizada exclusivamente para um projecto.

Figura 14.2: Janela de criação de um novo projecto.

Entre outros, o repositório inclui os seguintes ficheiros:

- Dois ficheiros dBase (extensão `dbf`), acompanhados pelos respectivos ficheiros índice, que representam duas tabelas relacionais que guardam meta-informação sobre os conceitos dos diagramas (classes, tabelas, atributos), e sobre as relações entre estes conceitos (por exemplo, quais os atributos pertencentes a cada classe).
- Dois ficheiros com a informação sobre cada diagrama do modelo, um com a parte da informação da apresentação gráfica (extensão `wmf`) do diagrama e outro com informação sobre o conteúdo (extensão `dgx`).
- Vários ficheiros para configuração do repositório e definição do comportamento da ferramenta, nomeadamente `usrprops.txt`, `saprops.cfg` e `sadeclar.cfg`.
- Um ficheiro (`network.lck`) que implementa mecanismos de *locking*, apenas relevantes nas instalações em ambiente de rede.

A utilização de tecnologia dBase, já relativamente ultrapassada, para armazenar informação sobre o conteúdo dos diagramas é ainda uma reminiscência da altura em que a ferramenta surgiu no mercado, em que este produto era líder na categoria de bases de dados individuais.

Adicionalmente, o facto destes ficheiros serem geridos directamente pelo sistema operativo, limita significativamente a possibilidade de utilização por equipas de trabalho de grande dimensão, e pode apresentar problemas de segurança e de integridade de informação, tal como já acontecia com o Rose. Por exemplo, qualquer utilizador pode apagar estes ficheiros, ou aceder e alterar dados, através de um programa, directamente nos ficheiros `dbf`.

Os ficheiros `usrprops.txt`, `saprops.cfg` e `sadeclar.cfg` são ficheiros de texto que definem um conjunto de parâmetros que controlam o comportamento da ferramenta. O primeiro é disponibilizado pelo System Architect para ser alterado pelo utilizador, de modo a acrescentar novas funcionalidades (ver exemplo na Secção 14.10). Os dois últimos são copiados quando se cria um novo projecto, a partir de uma cópia central de referência, e não devem ser alterados pelo utilizador. O seu conteúdo especifica as propriedades de todos os conceitos utilizados no System Architect, em função das abordagens seleccionadas. Como exemplo, veja-se a parte inicial do conteúdo do ficheiro `saprops.cfg`, relativamente ao conceito classe.

```
DEFINITION "Class"
  {
  #ifdef "Coad/Yourdon Object-oriented"

        CHAPTER "Attributes"
                      PROPERTY "Attributes"
                 { ZOOMABLE EDIT COMPLETE ListOf "Class Attribute"
                      KEYED BY {"Class Name":"Name", Name} ASGRID
                 LENGTH 4096 DISPLAY { FORMAT List } }
        CHAPTER "Services"
                      PROPERTY "Services"
                 { ZOOMABLE EDIT COMPLETE ParmListOf "Method"
```

```
                    KEYED BY {"Class Name":Name, "Formal Parameters"
QUALIFIABLE, Name}
                      ASGRID
                      LENGTH 2000 DISPLAY { FORMAT List } }
             CHAPTER "Coad/Yourdon"
                    GROUP ""
                    {
              LAYOUT { COLS 2 ALIGN OVER TAB }
                          PROPERTY "Notes"
                      { EDIT Text LENGTH 500 }
                          PROPERTY "additionalConstraints"
                      { EDIT Text LENGTH 500 }
                          PROPERTY "memoryRequirements"
                      { EDIT Numeric LENGTH 12 }
                          PROPERTY "timeRequirements"
                      { EDIT Text LENGTH 100 }
                    } REM "End of no name group"

                          PROPERTY "externalInput"
                      { ZOOMABLE EDIT ListOf "Data" LENGTH 2000 }
                          PROPERTY "externalOutput"
                      { ZOOMABLE EDIT ListOf "Data" LENGTH 2000 }
                          PROPERTY "Traceability Codes"
                      { ZOOMABLE EDIT ListOf "Requirement" LENGTH 2000 }

             CHAPTER "General"
```

O System Architect possibilita a exportação e importação de informação entre enciclopédias, de modo a permitir a reutilização de definições e de diagramas existentes. Neste caso, a ferramenta não assegura a manutenção da consistência entre as duas cópias de informação. Os modelos produzidos podem ser partilhados por diversos utilizadores e em vários projectos, neste caso recorrendo sempre aos já referidos mecanismos de importação e exportação.

O System Architect implementa algumas funcionalidades relacionadas com o repositório, recorrendo a ferramentas externas. Um exemplo disso é o controle de versões, realizado através de uma ligação a uma ferramenta especializada, designada PVCS (pode ser obtida mais informação sobre esta ferramenta em *www.merant.com*). Estão implementados diversos mecanismos de segurança, que permitem a definição de permissões, e a realização de operações de *check-in* e *check-out* de quaisquer objectos do repositório. Estas operações podem aplicar-se a um simples objecto ou a todo o modelo.

Num mesmo projecto, vários utilizadores podem aceder simultaneamente à mesma informação. Estão disponíveis mecanismos de *locking* ao nível de cada diagrama, definição, de todo o projecto, ou mesmo para o ficheiro que implementa o mecanismo de *locking* do projecto.

Todas as alterações efectuadas são registadas num *log* centralizado, que pode ser consultado. Podem ser definidas *passwords* por projecto, para as quais podem ser implementados mecanismos de *aging* (mudança obrigatória ao fim de um determinado período). O nível de segurança é estabelecido em função do conteúdo de uma biblioteca de ligação dinâmica (`sabind.dll`), que vem de origem em seis versões pré-configuradas distintas; o nível de segurança está directamente relacionado com a versão que estiver activa.

Finalmente, de referir que o System Architect tem disponível uma interface para o Microsoft Repository (ver Capítulo 12), através de um executável externo à aplicação. Estão disponíveis as operações de importação e exportação para o Microsoft Repository, o que possibilita a partilha de modelos com outras ferramentas de modelação. No entanto, e uma vez que a estratégia da Microsoft relativamente ao Microsoft Repository foi revista com o lançamento do SQL Server 2000, é natural que venham a surgir novidades nesta área por parte do System Architect no futuro próximo.

Ao nível de outros mecanismos de integração, de referir que o System Architect não tem qualquer suporte para XMI, pois não privilegia o UML em relação a outras notações.

14.4 Técnicas de Modelação

Ao contrário do que acontece com o Rose, que concentra a sua atenção nas notações de modelação do UML, o System Architect disponibiliza diversas outras técnicas e notações. Por isso, na Secção 14.5.1 são referidas as acções necessárias para definir as notações a utilizar num projecto, na Secção 14.5.2 analisaremos, de um modo geral, a forma como o System Architect integra as notações UML, enquanto na Secção 14.5.3 são apresentadas brevemente outras técnicas de modelação suportadas pelo System Architect.

14.4.1 Configuração das Propriedades do Projecto

Após a indicação da localização física do "repositório", e antes de terminar o processo de criação de um novo projecto, o System Architect solicita ao utilizador a especificação das técnicas de modelação que pretende aplicar no projecto, e que se encontram agrupadas nas seguintes categorias: (1) modelação de dados; (2) modelação do negócio; (3) métodos de modelação estruturados; (4) abordagens orientadas por objectos e (5) outras técnicas de modelação (incluem, por exemplo, desenho das interfaces das aplicações).

Dada a quantidade de opções disponibilizadas pela ferramenta, é aconselhável que apenas sejam seleccionadas as opções que de facto vão ser utilizadas. Caso contrário, pode-se tornar, por vezes, difícil identificar os parâmetros que devem ser preenchidos, quais os que podem ficar com valores por omissão, ou qual o impacto de se preencher ou não um parâmetro específico. Na Figura 14.3 podemos observar as notações de modelação disponíveis.

Nota: Na parametrização das notações a utilizar foram detectadas algumas incoerências, nomeadamente:

- Só foi possível seleccionar uma das notações de modelação orientada por objectos, ao contrário do que acontece com as notações estruturadas, em que todas puderam ser seleccionadas.
- A selecção de algumas notações estruturadas obrigou à inclusão prévia de outras, embora as condições não tenham sido sempre aplicadas coerentemente.

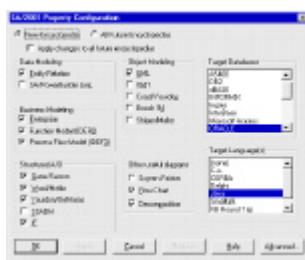



Figura 14.3: Janela de métodos disponíveis.

Esta parametrização inicial dos métodos a utilizar apresenta ainda um nível de detalhe superior, uma vez que se pode indicar, para os

métodos disponíveis, quais os diagramas que se pretende utilizar (ver Figura 14.4). Esta configuração pode ser alterada mais tarde, durante a realização do projecto.

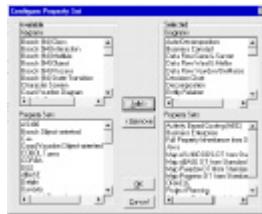



Figura 14.4: Diagramas disponíveis no System Architect.

14.4.2 O System Architect e o UML

O System Architect 2001 suporta a linguagem UML, com destaque para as seguintes características:

- Captura de requisitos e modelação de cenários através de casos de utilização.
- Modelação da dinâmica do sistema com os diagramas de sequência e de colaboração.
- Construção de diagramas de classes de forma iterativa, e modelação do comportamento das classes através dos diagramas de estado.
- Geração de código, para diversas linguagens, a partir dos diagramas de classes e recorrendo a um gerador de código configurável.
- Modelação do comportamento dos componentes.
- Modelação da instalação do software .

Na Figura 14.5, podemos observar os diagramas UML que o System Architect disponibiliza. Nesta categoria, podemos verificar que são incluídos alguns diagramas, que de facto não fazem parte desta linguagem de modelação, se bem que possam complementar a informação apresentada nos diagramas UML.

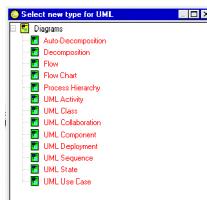



Figura 14.5: Diagramas UML suportados pelo System Architect

Para concretizar a implementação dos casos de utilização, o System Architect permite a representação de diagramas de sequência e diagramas de colaboração. Os dois diagramas anteriores permitem a representação da dinâmica do sistema, de acordo com o apresentado no Capítulo 7. O System Architect encarrega-se de manter, automaticamente, a consistência entre estes dois diagramas. Inclusivamente, pode-se optar por efectuar as alterações apenas num deles, que serão reflectidas no outro pelo System Architect.

O System Architect, não sendo uma ferramenta pura de modelação UML, possibilita algum nível de integração entre técnicas estruturadas e o UML. Por exemplo, existe uma forte integração com as técnicas de modelação de dados, estando disponíveis funcionalidades de geração de modelos de dados a partir dos diagramas de classes e vice-versa.

Estão também disponíveis algumas funcionalidades de integração entre os diagramas UML. Por exemplo, quando se coloca no diagrama de classes, uma classe com o mesmo nome que um actor de um diagrama de casos de utilização, o System Architect coloca essa classe no diagrama de classes como tendo um estereótipo actor. É também possível efectuar a ligação entre os métodos das classes e os eventos que constam dos diagramas de sequência.

14.4.3 Outras Técnicas de Modelação

O System Architect pretende ter uma resposta para a elaboração de diagramas de apoio a todas as fases do processo de desenvolvimento de software. Em particular, o System Architect é uma das ferramentas de referência na área da análise e desenho estruturado. São de destacar as seguintes funcionalidades de modelação nesta área:

- Suporte às notações estruturadas mais relevantes de diversos autores e/ou metodologias, nomeadamente: modelação de dados (Engenharia da Informação, Diagramas entidade associação); modelação de processos (abordagens de Gane e Sarson, Yourdon e DeMarco, SSADM IV); notações para problemas com restrições de tempo real (Ward e Mellor).
- Suporte a modelação de processos de negócio, com recurso às notações IDEF e às específicas do System Architect.
- Suporte a outras abordagens orientadas por objectos: Booch, Coad / Yourdon, OMT, Shlaer e Mellor.
- Modelação de dados, ao nível lógico e físico, com possibilidade de integração com os diagramas de classes do UML.
- Construção de matrizes CRUD (*Create, Retrieve, Update e Delete*, as quatro operações básicas sobre qualquer informação) para detalhar especificações de processos.
- Utilização e verificação das regras de diversas formas normais (Primeira, Segunda, Terceira e de Boyce-Codd).
- Geração e manutenção automática do esquema das bases de dados a partir do modelo de dados físico, com inclusão de especificidades de cada SGBD, nomeadamente: (1) a geração de instruções para criação de chaves primárias e secundárias; (2) instruções sobre regras, *stored procedures*, e valores por defeito; (3) *triggers* e *constraints*, relacionados ou não com a implementação da integridade referencial; (4) definições de índices.

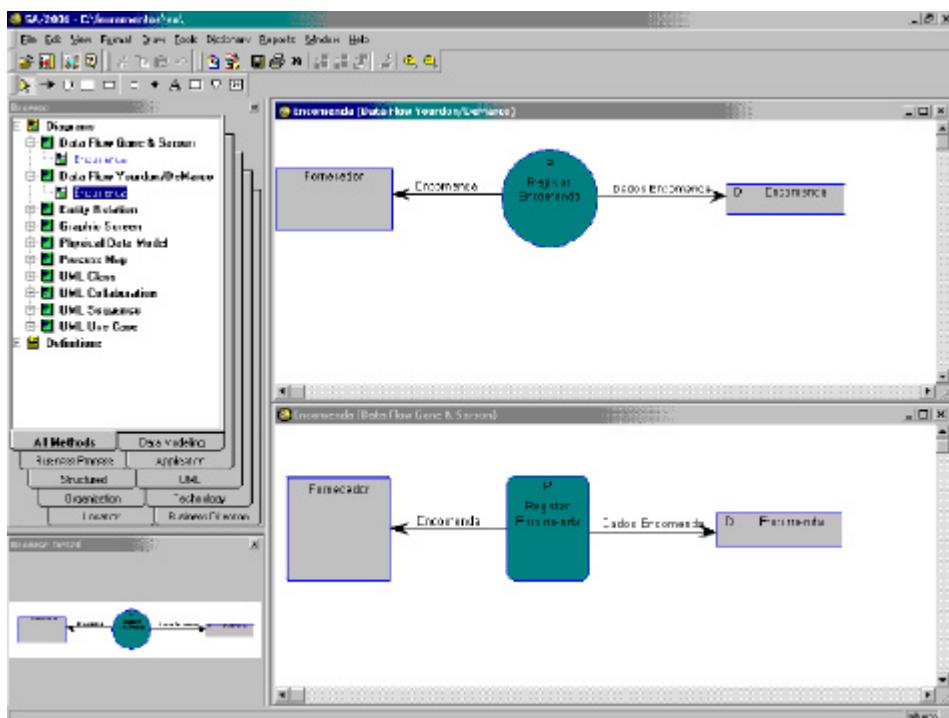



Figura 14.6: Exemplo de utilização de diagramas em notações diferentes para representar o mesmo processo.

O System Architect disponibiliza várias funcionalidades para efectuar a verificação da consistência da modelação efectuada num diagrama (por exemplo, verificação da existência de fluxos de dados incorrectos num DFD), mas, tal como a maioria das ferramentas, não o faz entre diagramas. A ausência destas validações possibilita que, num mesmo projecto, possam ser elaborados dois diagramas que representam a mesma informação, com o mesmo objectivo, mas que utilizam notações diferentes (Figura 14.6). Naturalmente, esta situação não deveria ocorrer.

14.5 Modelação do Negócio

O System Architect suporta um conjunto de notações e técnicas de modelação das funções do negócio e dos seus processos. Em particular, estão disponíveis os diagramas representados na Figura 14.7.

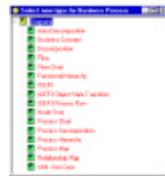



Figura 14.7: Diagramas de modelação do negócio suportados.

No caso da modelação do negócio, e como já discutido no Capítulo 12, o UML não suporta de base nenhuma notação específica para a sua representação, podendo ser definidas e aplicadas um conjunto de extensões de forma a ultrapassar esta deficiência.

O System Architect segue uma outra abordagem, disponibilizando um vasto conjunto de diagramas para modelar informação sobre o negócio, e que inclui, entre outros, os seguintes conceitos: (1) os objectivos do negócio; (2) a estrutura organizacional; (3) os processos de negócio segundo uma lógica de eventos; (4) as funções desempenhadas. O System Architect apresenta a possibilidade de captar muitos conceitos típicos do negócio, tais como factores críticos de sucesso, objectivos do negócio, informação sobre produtos, clientes, fornecedores, etc (ver Figura 14.8).

Para além de notações próprias, são ainda suportadas outras relativamente *standard*, nomeadamente os modelos IDEF (para mais detalhe consultar *www.idef.com*), e em particular o seu diagrama de funções do negócio. Estes diagramas são elaborados numa sequência hierárquica de detalhes sucessivos, desde um diagrama de contexto, em que toda a organização é representada num único processo, até ao nível de detalhe que for apropriado. São ainda utilizados diagramas em forma de árvore (*Node Tree Diagrams*), que permitem ver as relações directas entre um processo e respectivos sub-processos.

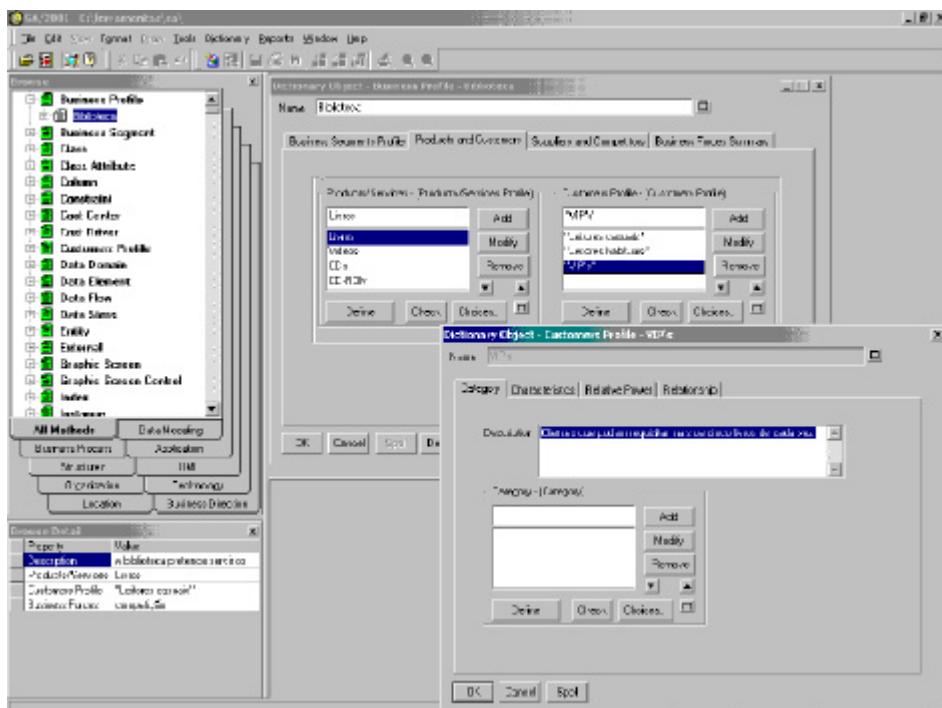



Figura 14.8: Conceitos de modelação de processos de negócio do System Architect..

Para além da representação de diagramas de processos de negócio, uma das funcionalidades tradicionais destas ferramentas é a disponibilização de opções de identificação de problemas, bem como de oportunidades de melhoria nos processos de negócio. Nesse sentido, o System Architect disponibiliza, entre outras, as seguintes funcionalidades:

- Ligação ao produto SimProcess Lite, um programa de simulação de processos de negócio da empresa CACI (*www.caci.com*), que é uma versão especificamente concebida para trabalhar com o System Architect.
- Análise de custos baseada nas actividades (*Activity Based Costing*).

A simulação, realizada pelo produto da empresa CACI, é suportada por uma ligação configurável. Este simulador permite avaliar o impacto de alterações do desenho dos processos, recorrendo à análise de métricas de desempenho, suportada por dados estatísticos.

14.6 Geração de Código – Caso de Estudo em Java

O System Architect disponibiliza, tal como o Rose, as funcionalidades de *forward e reverse engineering* a partir dos diagramas de classes de UML A partir de um diagrama de classes é também possível gerar um esquema para a criação de uma base de dados. O processo começa pela geração de um diagrama entidade associação a partir de um diagrama de classes, aplicando-se a partir desta fase técnicas estruturadas de modelação de dados.

Finalmente, o System Architect é das poucas ferramentas que disponibiliza o desenho e a correspondente geração das interfaces homem-máquina das aplicações, quer estas interfaces sejam de natureza gráfica ou baseadas em caracteres. Este é um mecanismo pouco comum nas actuais ferramentas de modelação CASE, como é o caso do Rose.

De referir que o System Architect tem um *wizard* para implementar um processo de *round trip engineering*, neste momento apenas disponível para o Visual Basic, mas cuja estrutura parece suficientemente genérica para no futuro suportar outras linguagens.

Nesta secção, vamos recorrer ao caso de estudo das "Encomendas e Fornecedores", já anteriormente referido e utilizado (ver Capítulo 13), para apresentar os vários mecanismos de geração de software do System Architect. Nesta secção apresentamos os mecanismos disponíveis para geração de código para a linguagem Java, e nas secções seguintes referiremos a geração de modelos de dados (Secção 14.8) e de interface homem-máquina (Secção 14.9).

14.6.1 Geração de Código

O System Architect permite a geração de código a partir de um diagrama de classes, segundo duas possibilidades:

- Recorrendo a um gerador de código interno, apenas disponível para as linguagens C++ ou Smalltalk.

- Utilizando um gerador de código que recorre às definições que constam de *scripts*, elaborados em linguagem VBA (*Visual Basic for Applications*), para outras linguagens e/ou ambientes de programação. São neste momento suportadas: C++, Java, Visual Basic, Object Pascal, Smalltalk, Powerbuilder ou CORBA IDL.

Para além destas linguagens, suportadas de base na ferramentas, existe a possibilidade de desenvolver *scripts* de geração de código para outras linguagens, utilizando o VBA no System Architect. Estes *scripts* encontram-se na subdirectoria `BasGen` da directoria do System Architect, e incluem o ficheiro `basgendc.inc`, que contem definições de funções de biblioteca do System Architect para aceder à informação dos objectos que se encontram no repositório. Estas primitivas estão documentadas, e portanto é possível, com algum trabalho, desenvolver um *script* específico.

A geração de código pelo System Architect é efectuada em menos passos, quando comparada com o Rose. Isto porque não existe a noção de componente associado a um programa/projecto de uma linguagem ou ambiente de desenvolvimento. Os passos que devem ser realizados passam pela:

- Definição das classes, incluindo os respectivos atributos, operações e relações entre elas.
- Selecção das classes para as quais se pretende gerar código.
- Selecção da linguagem para a qual se pretende gerar o código.
- Geração de código propriamente dita.

Definição de Classes

Tal como já referido, é através dos diagramas de classes que o System Architect, tal como a maioria das ferramentas, efectua a geração de código. Consideremos então o seguinte diagrama de classes (Figura 14.9), que modela os mesmos conceitos que foram apresentados no Capítulo do Rose. O mesmo diagrama poderia ser apresentado com as classes representadas pelo seu estereótipo (que, entre outros, pode assumir os valores *actor*, *boundary*, *control*, *entity*).

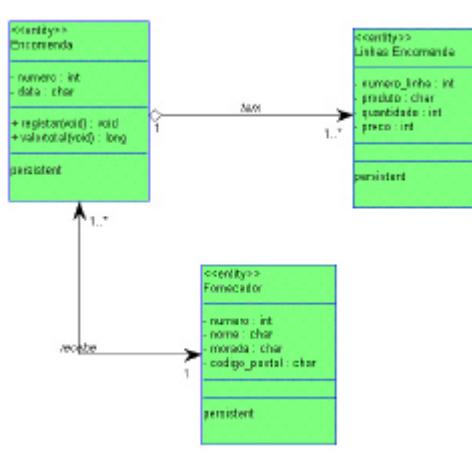



Figura 14.9: Diagrama de Classes.

Na definição de uma classe, é importante especificar algumas características gerais da classe, os seus atributos, as suas operações e as relações entre as várias classes do diagrama. Entre as características gerais, são de destacar as indicações sobre se a classe é persistente, abstracta e qual o seu estereótipo. Na Figura 14.10 podemos observar o aspecto do ecrã de especificação da classe `Encomenda`. Como se pode constatar, a informação a especificar para cada classe é muito extensa. Neste caso, e como apenas se pretendia efectuar a geração de código para Java, apenas aparece um tabulador relacionado com essa linguagem. Se tivéssemos seleccionado outras linguagens (C++, Visual Basic) para geração de código, estariam disponíveis outros tabuladores específicos para essas linguagens.

Figura 14.10: Ecrã de especificação da classe Encomenda.

Para cada atributo, para além da possibilidade de definir um conjunto de modificadores (por exemplo, se o atributo é estático), o tipo de dados e o tipo de acesso (e.g., *public*, *private*, *protected*), pode ainda ser especi-

ficada alguma informação relevante para a geração do modelo de dados (no tabulador, *Class, Description*), e que analisaremos mais à frente. Também o ecrã de especificação de um atributo apresentará um nível de complexidade maior se, adicionalmente, seleccionarmos outras linguagens para geração de código.

Figura 14.11: Ecrã de definição de um atributo.

Na Figura 14.11 podemos observar o ecrã de definição das propriedades do atributo `numero` da classe `Encomenda`.

Para cada método, e tal como para os atributos, é fundamental especificar um conjunto de modificadores (por exemplo, se o método é abstracto ou estático), o tipo de acesso (e.g., *public*, *private*, *protected*) e o tipo de dados que o método devolve. Para além desta informação, o System Architect permite especificar, para cada método, comentários, parâmetros formais e código, para serem incluídos no processo de geração de código.

Figura 14.12: Ecrã de definição do método registar.

Isto significa, à partida, que é possível manter a consistência entre o código e o modelo, pois ao contrário de muitas outras ferramentas, o System Architect permite a especificação de código no modelo e a importação de código de programas existentes. Na Figura 14.12 podemos observar o ecrã de definição das propriedades do método `registar` para a classe `Encomenda`.

Finalmente, e para completar a especificação da informação relacionada com uma classe, e para garantir a geração de código de forma

consistente, é ainda necessário efectuar a descrição de todas as relações entre classes. No caso do nosso exemplo, temos duas relações de associação, e a especificação da relação entre as classes `Encomenda` e `Fornecedor` pode ser observada na Figura 14.13.

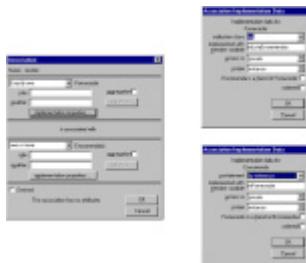



Figura 14.13: Definição da relação entre as classes Fornecedor e Encomenda.

No ecrã da esquerda, é especificada a multiplicidade de cada um dos intervenientes na relação, e se a relação é de agregação ou não. Para que esta associação seja representada no código gerado a partir do diagrama, é necessário preencher informação sobre as *Implementation Properties,* nas classes da relação que se pretende que tenham referências para a outra participante. Neste caso, optámos por incluir uma referência em ambas as classes, e é essa informação que é apresentada nos ecrãs do lado direito:

- O do canto superior direito especifica as propriedades da referência a incluir na classe `Fornecedor` para a classe `Encomenda`: um fornecedor poderá receber várias encomendas, por isso a relação é concretizada por uma lista, que na classe `Fornecedor` será representada por uma variável de instância privada, designada `mListaEncomendas`.
- O ecrã inferior especifica as propriedades da referência a incluir na classe `Encomenda` para a classe `Fornecedor`: como uma encomenda diz respeito apenas a um fornecedor, por isso a relação é concretizada por uma variável de instância privada `mFornecedor`.

No caso do UML, o System Architect disponibiliza uma funcionalidade, na opção *Reports - Unified Rules Check* (apenas aplicável quando é apresentado um diagrama UML), que permite verificar, antes da

geração de código, se os diagramas estão de acordo com as regras do UML e da semântica da linguagem de programação seleccionada para geração. Os erros e os avisos são assinalados no diagrama, conjuntamente com uma janela onde aparece a respectiva descrição. No caso do diagrama de classes apresentado na Figura 149, o conteúdo desta janela é o apresentado na Figura 14.14.

Figura 14.14: Relatório de validação do diagrama de classes das Encomendas.

Geração de Código

Esta geração é realizada através da opção *Dictionary - Forward and Reverse Code Engineer*. Após a apresentação de uma nova janela com um menu, é necessário efectuar vários passos: (1) seleccionar todas as classes para geração de código (opção *Selection - Select All*); (2) seleccionar o *script* a utilizar durante o processo de geração para Java (opção *Options - Script Gen*); (3) gerar o código (opção *File - Export Via Script*).

O resultado obtido é colocado num ficheiro, cujo nome é parametrizável, e é criado um ficheiro de *log* com as alterações realizadas.

De acordo com a especificação realizada no diagrama de classes apresentado anteriormente, e tendo em conta que a linguagem seleccionada é Java, apresenta-se de seguida o código gerado para todas as classes do diagrama.

```
/* saOutput.java */
/* Generated by System Architect Tue Mar 27 12:10:10 2001 */

package compras;

/*#SABEGIN
       Class Name = Linhas Encomenda,
       #SAEND*/
class Linhas_Encomenda
{
```

```
    /*#SABEGIN
       Attribute Name = numero_linha,
       ,
       #SAEND*/
    private int numero_linha;
    /*#SABEGIN
       Attribute Name = produto,
       ,
       #SAEND*/
    private char produto;
    /*#SABEGIN
       Attribute Name = quantidade,
       ,
       #SAEND*/
    private int quantidade;
    /*#SABEGIN
       Attribute Name = preco,
       ,
       #SAEND*/
    private int preco;

    /*#SABEGIN Member Name = mEncomenda, Access Code = private,
    Is Aggregation = N, Association type = by reference,
    Scope = , Data Type = Encomenda&,
    CollectionOf = , Cardinality = 3,
    Other Cardinality = 1, Other Class = Encomenda,
    Association = tem, #SAEND*/

    private Encomenda mEncomenda;

}

class Fornecedor
{
    /*#SABEGIN
       Attribute Name = numero,
       Primary Key = Y,
       #SAEND*/
    private int numero;
    /*#SABEGIN
       Attribute Name = nome,
       ,
       #SAEND*/
    private char nome;
    /*#SABEGIN
       Attribute Name = morada,
       ,
       #SAEND*/
    private char morada;
    /*#SABEGIN
```

```
           Attribute Name = codigo_postal,
           ,
           #SAEND*/
        private char codigo_postal;

        /*#SABEGIN Member Name=mListaEncomenda, Access Code = private,
        Is Aggregation = N, Association type = by reference,
        Scope = , Data Type = CObList,
        CollectionOf = , Cardinality = 1,
        Other Cardinality = 3, Other Class = Encomenda,
        Association = recebe, #SAEND*/

        private CObList[] mListaEncomenda;

    }

    class Encomenda
    {
        /*#SABEGIN
           Attribute Name = numero,
           Primary Key = Y,
           #SAEND*/
        private int numero;
        /*#SABEGIN
           Attribute Name = data,
           ,
           #SAEND*/
        private char data;

        /*#SABEGIN Member Name=mLinhasEncomenda, Access Code= private,
        Is Aggregation = Y, Association type = by reference,
        Scope = , Data Type = CObList,
        CollectionOf = , Cardinality = 1,
        Other Cardinality = 3, Other Class = Linhas Encomenda,
        Association = tem, #SAEND*/

        private CObList[] mLinhasEncomenda;

        /*#SABEGIN Member Name = mFornecedor, Access Code = private,
        Is Aggregation = N, Association type = by reference,
        Scope = , Data Type = Fornecedor&,
        CollectionOf = , Cardinality = 3,
        Other Cardinality = 1, Other Class = Fornecedor,
        Association = recebe, #SAEND*/

        private Fornecedor mFornecedor;

        public void registar()
        {}
        public long valortotal()
        {}
    }
```

Para além do código propriamente dito, o System Architect gera um conjunto de informação adicional, que especifica propriedades importantes das classes, e que permitem garantir a consistência entre este código e o modelo. Muita desta informação está relacionada com os atributos e as relações. No entanto, e ao contrário do Rose, o System Architect não associa um identificador único cada item do código gerado (classe, atributo, método), o que muitos autores apontam como factor negativo no sentido de garantir a integridade entre o modelo e o código. A análise do código gerado neste caso aponta para que globalmente, os resultados esperados foram os obtidos.

No caso da linguagem Java, a geração de relações de herança é também realizada adequadamente pelo System Architect, através da cláusula *extends* colocada no código da definição da subclasse.

14.6.2 Reverse Engineering

O System Architect suporta *reverse engineering* de C++ e de Java, e em muitas situações, podemos constatar que o processo é adequadamente realizado.

Assim, como exemplo, consideremos a adição de um novo método ao código gerado para a classe `Linhas_Encomenda`, para calcular o valor total da linha, e com o seguinte aspecto.

```
public long valortotal()
{ return( quantidade * preco ) ; }
```

A realização do processo de *reverse engineering* cria correctamente um novo método na classe `Linhas_Encomenda`, coloca o tipo de acesso como *public* e o tipo do valor devolvido como *long* e ainda coloca no modelo o código introduzido para o método. Se, de seguida alterássemos o código directamente no modelo, esta alteração seria reproduzida no código na próxima geração a realizar. É assim possível manter o modelo e o código consistentes, o que é condição necessária e sufici-

ente para a definição e implementação correcta de um mecanismo de *round-trip engineering*.

Outras alterações efectuadas directamente no código Java foram repercutidas correctamente no modelo, nomeadamente:

- Alteração do tipo de dados e do tipo de acesso.
- Adição de uma nova classe, subclasse de outra já existente. Não só a classe foi criada correctamente, como a relação entre as duas classes foi definida no modelo.

De uma forma geral, o System Architect suporta adequadamente os processos de *forward engineering* e de *reverse engineering* na linguagem Java, se considerarmos as situações mais normais e a utilização típica deste tipo de funcionalidades.

Existem contudo algumas situações que o System Architect não traduziu correctamente no modelo, mas cuja ocorrência também será menos frequente. Por exemplo, no caso de uma relação de associação definida no modelo, esta é correctamente expressa no código gerado. No entanto, se depois a relação de associação for directamente alterada no código para herança, eliminando do código qualquer referência à relação de associação, a importação do código para o modelo deixa-os inconsistentes, pois no modelo ficam definidas quer a relação de associação quer a de herança.

14.7 Geração de Modelos de Dados

O System Architect realiza a geração de esquemas de criação / alteração da estrutura de bases de dados, sempre a partir de diagramas entidade associação. Não existe qualquer relação directa entre os diagramas UML e os esquemas de bases de dados. De forma a ultrapassar este facto, o System Architect possibilita a geração de um diagrama entidade associação de nível lógico, a partir de um diagrama de classes em UML, e a partir daqui são utilizadas técnicas de modelação estruturadas para se obter um *script* com a estrutura de bases de dados. Através desta estratégia, são suportadas inúmeras bases de dados, nomeadamente: AS/400, ANSI SQL, dBase III, DB2, Informix, Interbase, Oracle, Paradox, Progress, Rdb, SqlBase, Sql Server, Sybase, XDB, Teradata.

Estas técnicas possibilitam também a realização do processo inverso, ou seja, de *reverse engineering* de uma base de dados: a partir de um *script* existente, é elaborado um diagrama entidade associação de nível físico, a partir do qual se pode construir o diagrama entidade associação de nível lógico. A partir deste é possível criar um diagrama de classes UML.

Constata-se que o System Architect tira partido da sua larga experiência e do suporte de técnicas estruturadas de modelação de dados, integrando-as com os diagramas de classes. As primeiras são técnicas maduras relativamente à geração de esquemas de bases de dados, quando comparadas com as abordagens orientadas por objectos.

Assim, o processo de geração de um *script* SQL a partir dos diagramas de classes passa por:

- Definição de *Data Elements,* que no fundo não são mais do que especificações dos tipos de dados que pretendemos atribuir aos diversos atributos de uma classe, o que implica uma especificação prévia. São os tipos de dados especificados por estes *Data Elements* que são utilizados na geração dos modelos de entidade associação, e não os tipos que observamos nos diagramas de classes.
- Definição de um diagrama de classes com as classes que pretendemos mapear em tabelas da base de dados.
- Geração automática de um diagrama entidade associação de nível lógico, a partir do diagrama de classes anterior.
- Realização de eventuais alterações ao nível deste diagrama.
- Geração de um diagrama entidade associação de nível físico e realização de eventuais alterações a este diagrama.
- Geração do *script* de criação da base de dados.

Todos os conceitos mais próximos das bases de dados (*constraints*, *stored procedures*, *triggers*) são considerados ao nível dos modelos entidade associação, o mesmo acontecendo com a verificação de diversas formas normais (primeira, segunda, terceira e *Boyce-Codd*), para garantir a conformidade com as regras do modelo relacional e a adição ou remoção de chaves estrangeiras.

De forma a exemplificarmos a forma como o System Architect realiza o processo de geração de um *script* de criação de bases de dados a partir de um diagrama de classes, utilizemos o mesmo exemplo que no caso do Rational Rose (Figura 14.15).

O tabulador *Entity Info* permite especificar alguma informação sobre o comportamento da classe sempre que for utilizada num processo de geração de um modelo de dados, sendo possível especificar, entre outros, o nome da tabela e um prefixo a aplicar a aos nomes de todas as colunas.

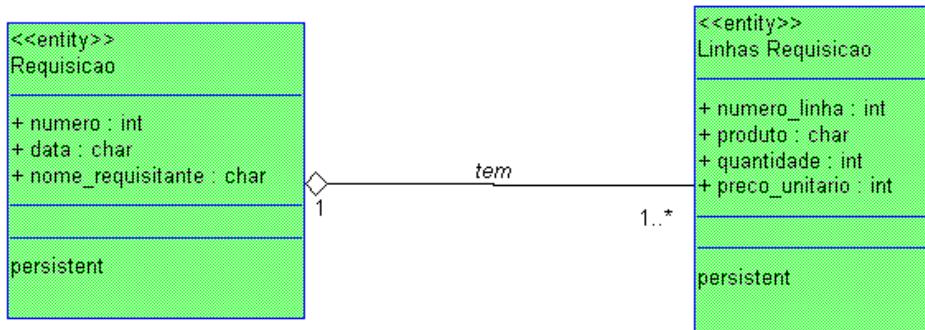



Figura 14.15: Diagrama de classes de Requisições.

A definição dos *Data Elements*, disponíveis na categoria *Data Modeling*, é efectuada de forma global a todo o projecto. A cada um destes elementos podem ser atribuídos um domínio de dados. Os valores a definir para um *Data Element* podem ser observados na Figura 14.16.

Figura 14.16: Definição de um Data Element.

Depois de efectuada a especificação destes tipos de dados, podemos sucessivamente: (1) gerar o diagrama entidade associação de nível

lógico (Figura 14.17); (2) gerar o diagrama entidade associação de nível físico (Figura 14.18); (3) gerar o *script* de criação das tabelas.

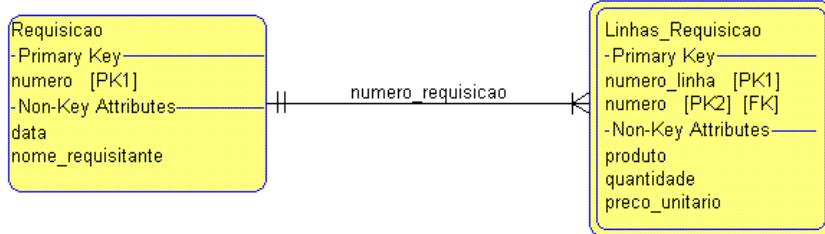



Figura 14.17: Diagrama entidade associação de nível lógico.

No diagrama entidade associação de nível lógico podemos definir alguns conceitos típicos das bases de dados, como *constraints* e *triggers*. Se não efectuarmos qualquer alteração, o aspecto do diagrama de nível físico gerado é o apresentado na Figura 14.18. Para a geração correcta de esquemas de bases de dados, é necessário preencher adequadamente alguns parâmetros ou executar algumas opções. Por exemplo, é importante indicar na relação entre as entidades `Requisicao` e a `Linhas_Requisicao` que o atributo chave primária da primeira também identifica a segunda, de modo a ser colocado como componente da chave primária desta última.

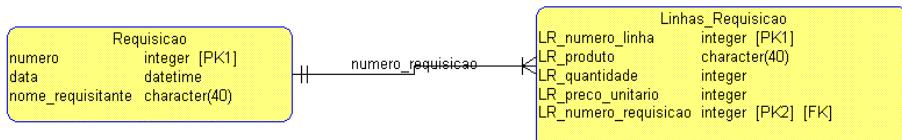



Figura 14.18: Diagrama entidade associação de nível físico.

O código gerado para o *script* de criação das tabelas anteriores, no caso de uma base de dados SQL Server, é o seguinte:

```
CREATE TABLE Linhas_Requisicao(
          LR_numero_linha               integer NOT NULL,
          LR_produto                   character(40) NOT NULL,
          LR_quantidade                 integer NOT NULL,
          LR_preco_unitario             integer NOT NULL,
          numero                        integer NOT NULL)
go

ALTER TABLE Linhas_Requisicao ADD CONSTRAINT Linhas_Requisicao_PK
    PRIMARY KEY  (LR_numero_linha,numero)
go

CREATE TABLE Requisicao
         ( numero                       integer NOT NULL,
          data                          datetime NOT NULL,
          nome_requisitante            character(40) NOT NULL)
go
```

O System Architect apresenta várias funcionalidades relacionadas com a geração de esquemas de bases de dados. Por exemplo, na fase de geração do esquema é possível indicar se o objectivo é gerar apenas os comandos de criação de tabelas ou também os relacionados com a criação de chaves primárias ou estrangeiras, ou mesmo com a criação de *triggers* que implementam a integridade referencial. A explicação detalhada de todas estas opções encontra-se fora do âmbito deste livro.

Relações de Generalização

A ferramenta concretiza adequadamente a geração de um diagrama entidade associação para o caso de uma relação de generalização.

Por exemplo, se considerarmos o diagrama de classes da Figura 14.19, em que `classe2` é uma subclasse de `classe1`, o diagrama entidade associação gerado apresenta o aspecto ilustrado na Figura 14.20.

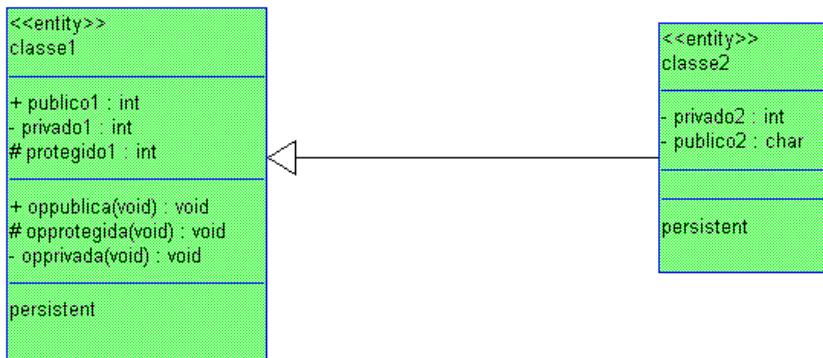
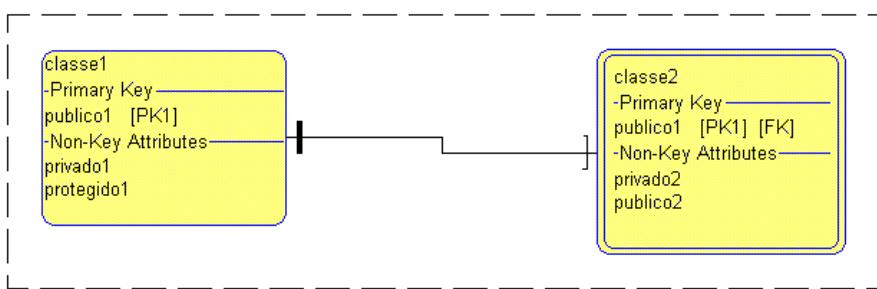



Figura 14.19: Diagrama de classes representando uma relação de generalização.

Como podemos observar na Figura 14.20, o diagrama entidade associação gerado inclui na tabela correspondente à classe `classe2` um atributo `publico1`, que representa a chave primária da tabela `classe1`, e concretiza a relação de generalização ao nível do modelo de dados. Esta é a forma correcta de concretizar este tipo de relações, sendo que a especificação da relação, no diagrama entidade associação, inclui informação que permite facilmente identificá-la como tendo sido originada por uma relação de generalização de um diagrama de classes (esta identificação pode também ser efectuada visualmente através dos simbolos utlizados para a representação da relação). A partir deste diagrama seriam utilizadas técnicas estruturadas para se obter o esquema da base de dados.

Figura 14.20: Diagrama entidade associação resultante de relação de generalização.

Reverse Engineering

A operação de *reverse engineering* está disponível, a partir da própria estrutura de uma base de dados, mas sempre passando por um diagrama entidade associação, a partir do qual é possível gerar um diagrama de classes.

14.8 Geração de Interfaces Homem-Máquina

Uma das áreas em que tradicionalmente as ferramentas de modelação apresentam mais lacunas é na especificação das interfaces homem-máquina das aplicações, de forma integrada com a lógica de processamento. No passado, este problema poderia não ser demasiado grave, pois o próprio código do programa era normalmente responsável pela definição das interfaces. Apenas as ferramentas especificamente vocacionadas para o desenho do sistema (*Lower Cases*) incluíam algum suporte para a geração de interfaces, mas muitas vezes tal era (e é) feito de forma desintegrada com a especificação da lógica de processamento. Com o aparecimento das interfaces gráficas, esta questão ganhou maior relevância.

O System Architect disponibiliza diversas funcionalidades para o desenho de interfaces, gráficas ou orientadas ao caracter. O *Screen Painter* permite desenhar ecrãs para ambientes Windows, designadamente para aplicações Visual Basic, Powerbuilder ou Delphi, e também para aplicações modo caracter (em Cobol). Por exemplo, é possível

gerar automaticamente os ficheiros do Visual Basic que contêm o desenho da interface, com a localização e dimensão dos diversos objectos, incluindo menus e opções.

Os ecrãs são implementados através de diagramas definidos na categoria *Application*, como se pode observar na Figura 14.21.

Figura 14.21: Diagramas disponíveis na categoria Application.

Estes diagramas não estão directamente relacionados com nenhum conceito do UML, a não ser pela possibilidade de alguns elementos da interface serem associados a definições que constam de outros diagramas do projecto (por exemplo, atributos de classes). É possível indicar, para cada componente visual colocado num ecrã, uma relação com outro conceito que integra o modelo, utilizando a propriedade *Deriving Element* da especificação do componente. Por exemplo, se quiséssemos dizer que a *Textbox* do `Fornecedor` estava relacionada com o atributo nome da classe `fornecedor`, deveríamos colocar o seguinte conteúdo na referida propriedade: "nome" / "Class" ""Fornecedor""/.

Os ecrãs podem ser criados recorrendo a um conjunto de diferentes técnicas, designadamente:

- Podem ser desenhados pela colocação num ecrã de diversos controles disponibilizados pelo System Architect., nomeadamente os mais comuns na interface de programas Windows: *textboxes*, *labels* (textos estáticos), *comboboxes*, *listboxes*, *radiobuttons*, *checkboxes*, *push buttons*, etc.
- Podem ser gerados a partir de outras definições existentes noutros diagramas do System Architect. Por exemplo, é possível seleccionar uma classe de um diagrama de classes e por mecanismos de *drag-and-drop*, colocá-la num diagrama "graphic screen"; o System Architect gera automaticamente um ecrã, onde são representados todos os atributos da classe.

Na Figura 14.22, podemos observar a concretização desta última possibilidade: o ecrã do lado direito em baixo representa um diagrama de classes; o ecrã do lado direito em cima é o ecrã gráfico gerado a partir do diagrama de classes. Podemos ainda observar ao lado do ecrã gráfico uma janela de definição das propriedades do controle `numero` e constatar que o conteúdo da propriedade *Deriving Element* foi automaticamente preenchido, de forma a associar esse campo a um atributo de um classe. Este ecrã ilustra ainda a possibilidade de construir ecrãs com mais do que uma tabela, inclusivamente do tipo *master-detail*.

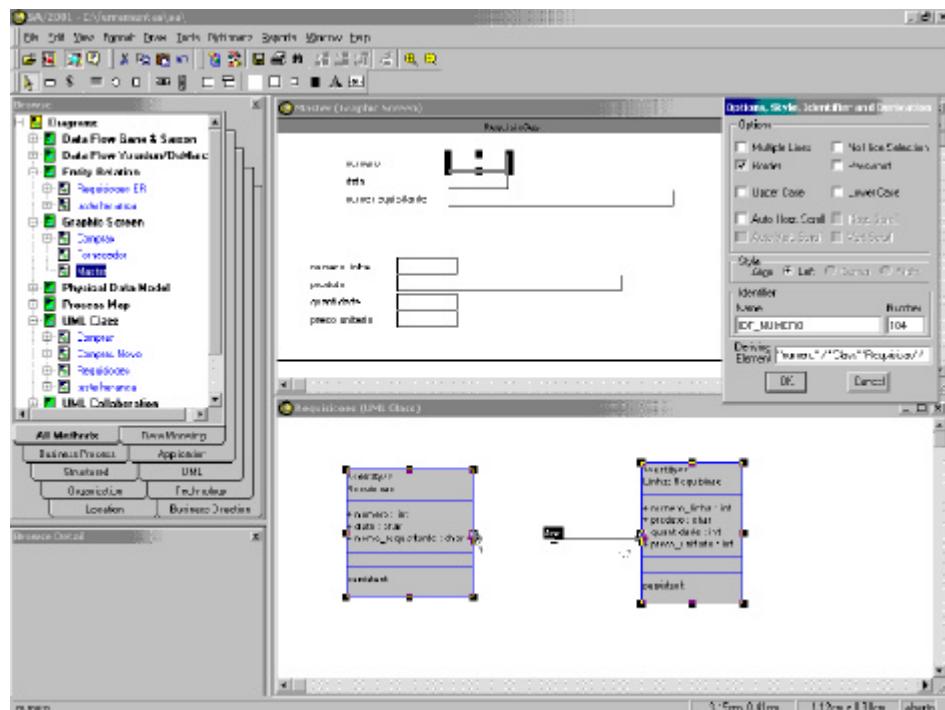



Figura 14.22: Exemplo da construção de um ecrã de interface.

Naturalmente, coloca-se o problema de integração entre esta interface e o código que eventualmente se pretenda associar à mesma, o que não é possível de efectuar.

14.9 Mecanismos de Extensibilidade

O System Architect é uma ferramenta que, tal como o Rose, recorre a diversos mecanismos para disponibilizar capacidades de extensão relativamente às suas características base. Elas podem ser agrupadas nas seguintes categorias:

- Suporte do VBA (*Visual Basic for Applications*), como linguagem para a elaboração de *scripts*, que podem ser utilizados para diversas funcionalidades, nomeadamente automatização de tarefas e produção de relatórios.
- Disponibilização de uma interface de programação (API) para acesso ao conteúdo do repositório, nomeadamente a todas as definições e respectivas propriedades. Esta API encontra-se documentada, pelo que é possível a sua utilização em *scripts* desenvolvidos pelo utilizador.
- Possibilidade de modificação de propriedades da ferramenta, alterando o conteúdo do ficheiro `usrprops.txt`. Este ficheiro permite alterar e/ou acrescentar características definidas no ficheiro `saprops.cfg`, fornecido de base pelo System Architect.
- Possibilidade de referenciar nalguns ambientes de programação, nomeadamente em Visual Basic, bibliotecas dinâmicas que disponibilizam acesso directo às propriedades e informação que se encontra no repositório.

O VBA pode também ser utilizado no desenvolvimento de funcionalidades de integração com outras ferramentas (para além das funcionalidades já existentes, que utilizam o Microsoft Repository, ou opções específicas), possibilitando a importação e exportação de informação gerida pelo repositório, através de *scripts*.

De forma a exemplificar alguns dos mecanismos de extensibilidade atrás referenciados, vamos considerar uma situação hipotética em que se pretende recolher informação adicional sobre qualquer classe, e incluí-la no código gerado. Em primeiro lugar, é necessário alterar o conteúdo do ficheiro `usrprops.txt`, de modo a alterar a informação que é solicitada a propósito de cada classe. A estrutura deste ficheiro encontra-se documentada nos manuais e na ajuda da ferramenta, mas

é bastante simples e facilmente compreensível, como se ilustra no exemplo seguinte.

A alteração efectuada neste ficheiro pode ser interpretada da seguinte forma: "Na definição de qualquer classe, incluir uma nova propriedade, designada `Comentário Adicional Classe`, cujos valores serão introduzidos e alterados numa *Textbox*, com tamanho máximo de 60 caracteres."

```
	REM "USRPROPS.TXT"
	REM "Copyright Popkin Software. All rights reserved."
	REM "Instructions for modifying this file are in the on-line
help."

	Definition "Class"
	{
	PROPERTY "Comentário Adicional Classe"
	{EDIT Text LENGTH 60}
	}
```

O impacto desta alteração pode ser visualizado na Figura 14.23, em que no ecrã de especificação da classe `Encomenda` se pode observar um novo campo.

Figura 14.23: Inclusão de novo campo na especificação de qualquer classe.

O conteúdo deste novo campo passa a ficar guardado no repositório, para cada classe, o que demonstra a capacidade de expansão da estrutura do repositório. Como o objectivo é reflectir o conteúdo deste novo campo no código Java gerado, é necessário modificar o *script* de

geração de código desta linguagem. Para isso, é necessário incluir o seguinte código no ficheiro `java.bas`.

```
'get Comentário Adicional Classe
a$ = BasGenGetProp(classId&,"Comentário Adicional Classe")
' imprimir Comentário Adicional Classe em comentário
if ( a$ <> "" ) then
       print "/* ",a$," */"
end if
print " "
comment$ = BasGenGetProp(classId&,"Java Class Comment Text")
```

De realçar a chamada à função `BasGenGetProp`, que o System Architect disponibiliza para determinar o valor que uma "propriedade" tem para uma determinada definição, neste caso identificada por `classId`. Numa próxima geração de código, o código gerado apresentaria como principal diferença a inclusão do texto introduzido no campo da Figura 14.23 imediatamente antes da código da classe `Encomenda`., como podemos constatar no exemplo seguinte.

```
/* saout.java */
/* Generated by System Architect Tue Feb 27 16:45:36 2001 */
package compras;
/* Teste Comentário Adicional */
class Encomenda
{
…..
```

14.10 Geração de Documentação

O System Architect disponibiliza três funcionalidades principais para a elaboração de documentação:

- Um sistema interno de *Reporting*, baseado na utilização de uma linguagem semelhante ao SQL para acesso ao repositório.

- Uma ligação ao Microsoft Word, para a elaboração de um conjunto muito vasto de relatórios pré-definidos. Estes *templates* permitem também o acesso a informação do repositório, e podem ser alterados, uma vez que foram desenvolvidos em VBA. O utilizador pode utilizar a mesma técnica para a elaboração de novos relatórios.
- Um gerador de relatórios HTML, que possibilita a geração de páginas HTML formatadas, incluindo os diagramas e definições de um projecto, aos quais se adicionam funcionalidades de navegação típicas de um *browser*.

Adicionalmente, o System Architect possibilita a elaboração de relatórios diversos, que podem aceder ao conteúdo do repositório, através do desenvolvimento de *scripts* VBA.

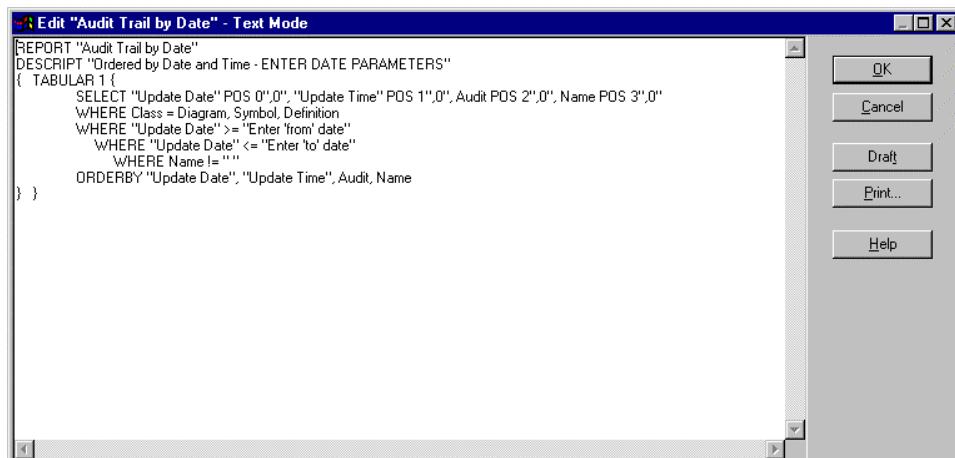



Figura 14.24: Editor de relatórios internos.

O sistema interno de *reporting*, ao qual o utilizador pode também recorrer, é muito intuitivo e fácil de utilizar. Na Figura 14.24, podemos observar o aspecto do editor onde é definido o código deste tipo de relatórios, bem como o exemplo de um relatório cujo objectivo é apresentar informação sobre as alterações efectuadas ao modelo.

Para além deste editor de relatórios, em que se manipula directamente o código, o System Architect possibilita a construção dos mesmos relatórios através de uma interface em que o utilizador especifica a informação que pretende obter do repositório, sem ter que introduzir

código e saber a sintaxe da linguagem de elaboração de relatórios (ver Figura 14.25).

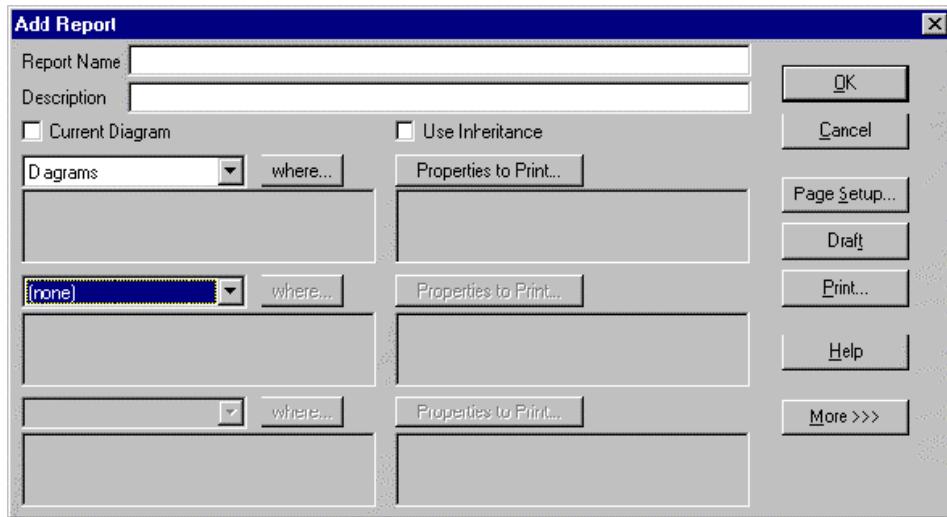



Figura 14.25: Editor gráfico de relatórios internos.

Dado o número elevado de relatórios pré-definidos para Word, o System Architect disponibiliza *templates* que agrupam os relatórios em categorias, em função do tipo de técnica de modelação a que dizem respeito (por exemplo, modelação de dados, de objectos, do negócio, estruturadas).

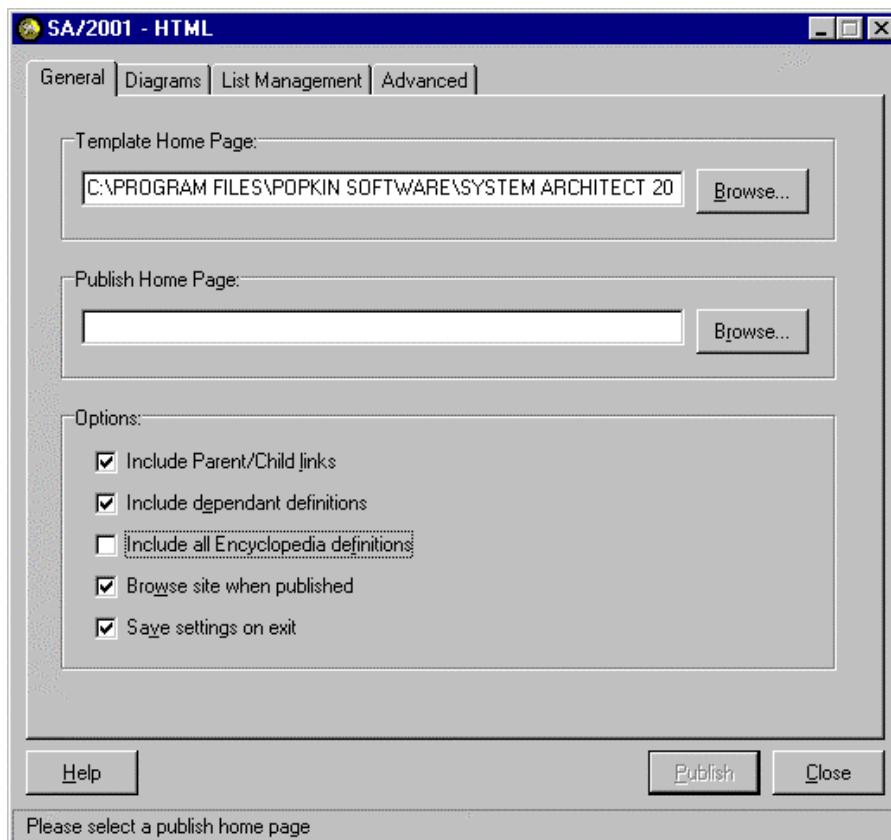



Figura 14.26: Gerador de páginas HTML.

Finalmente, o gerador de páginas HTML permite criar um conjunto de páginas com informação relacionada e que consta do repositório, sendo possível especificar quais os diagramas que se pretende incluir no âmbito da geração. Esta especificação mais detalhada é realizada no tabulador *Diagrams* da janela apresentada na Figura 14.26.

14.11 Conclusão

O System Architect é uma das mais antigas ferramentas de modelação de software, e que ainda continua a manter uma posição de destaque. Apareceu inicialmente no mercado como ferramenta de modelação de software, recorrendo a técnicas estruturadas, quer de modelação de processos quer de dados. Ao longo do tempo, o System Architect foi evoluindo e incorporando outras notações (modelação do negócio e

orientadas por objectos). As suas últimas versões passaram a incluir também a linguagem UML, sem nunca deixar de suportar as anteriores notações.

Ao concluir este capítulo, gostaríamos de efectuar uma análise comparativa, se bem que resumida, entre as duas ferramentas. Não faz sentido comparar a utilização do UML por cada uma das ferramentas; ambas suportam adequadamente as regras desta linguagem de modelação, se bem que o Rose utiliza-a de forma quase exclusiva, enquanto que para o System Architect o UML é mais uma notação de modelação, sem nenhum privilégio especial. Aliás, a abrangência do System Architect fica expressa não só neste facto, como também por outros factores, por exemplo, o elevado número de sistemas de gestão de bases de dados para os quais suporta a geração de esquemas.

Os factores que diferenciam as ferramentas de modelação em UML são cada vez menos questões relacionadas especificamente com a linguagem UML, mas sobretudo com funcionalidades que permitem a verificação da consistência dos diagramas, a geração automática de outros artefactos, a partir dos diagramas UML, ou a possibilidade de adicionar novas funcionalidades à ferramenta.

Neste aspecto, o System Architect tira algum partido da experiência acumulada ao longo dos anos, pois resolveu uma situação não completamente solucionada pelo UML - o mapeamento de modelos de classes em esquemas das bases de dados - recorrendo a passos intermédios, que implicam a utilização de técnicas estruturadas (diagramas entidade associação), com provas dadas ao nível de geração de esquemas das bases de dados. Reconhecendo a incapacidade de suporte adequado ao nível do UML, e não podendo tirar partido de técnicas estruturadas que não suporta, o Rose optou por, na versão Rose 2001, definir um novo tipo de diagramas e de conceitos, que aproximam mais as técnicas de modelação em UML dos conceitos das bases de dados (ver Capítulo 13).

Noutras áreas analisadas, o Rose aparenta possuir alguma vantagem, nomeadamente ao nível dos mecanismos de extensibilidade disponibilizados, uma vez que os mecanismos existentes no caso do Rose estão mais integrados com a linguagem UML (por exemplo, ao nível das clas-

ses existentes na REI), quando comparados com os do System Architect, que são aplicáveis a qualquer tipo de notação por ele suportada.

Finalmente, de referir que a evolução do Rose tem decorrido a um ritmo superior à do System Architect (o que é compreensível, pois este tem que continuar a suportar múltiplas técnicas, por compatibilidade com versões anteriores, e porque é essa a sua estratégia), o que nos permite concluir que o Rose se apresenta com vantagens naturais para quem pretenda concentrar a sua atenção na modelação de software em UML.

Apêndices, Bibliografia e Índice Remissivo

APÊNDICE A – GUIA DE RECURSOS ELECTRÓNICOS

Neste apêndice é compilado um conjunto significativo de referências electrónicas (links) relacionadas com os temas abordados neste livro. Esta compilação está organizada num conjunto de categorias, de modo a facilitar ao leitor a procura e navegação de informação. As categorias consideradas são as seguintes:

- ***Standards* e Organizações Normalizadoras**: São tipicamente consórcios de empresas e de centros de investigação cujo o objectivo é a promoção, divulgação de informação sobre os métodos de A&D, linguagens de modelação, e processos de desenvolvimento de software, ou então referências disponíveis online sobre algumas normas referidas ao longo do texto.
- **Leituras Recomendadas**: Constituem um conjunto de referências clássicas, suficientemente simples de entender, e que abordam diversos aspectos que poderão facilitar a compreensão dos temas abordados.
- **Catálogos de Informação (Meta-Informação)**: Estes recursos constituem excelentes pontos de partida para uma procura geral sobre os temas abordados; apresentam um número significativo de referências electrónicas para distintos locais, organizados por distintos critérios; alguns destes recursos providenciam ainda jornais online, grupos de discussão e/ou listas de correio-electrónico. Poderão incluir referências para recursos bibliográficos existentes na Internet, nomeadamente para livros, revistas, artigos publicados em conferências, etc.

- **Empresas**: Lista de empresas cuja actuação tem, ao longo do tempo, apresentado um significado relevante para os temas abordados neste livro. Algumas empresas não são incluídas nesta lista pelo facto de serem referidas no ponto seguinte.
- **Ferramentas CASE**: Lista de empresas e ferramantas de modelização de software. Esta lista, sem pretender ser exaustiva, apresenta algumas das referências mais relevantes nesta categoria de ferramentas, podendo servir como ponto de partida para o leitor efectuar a sua própria investigação.

Standards, Organizações Normalizadoras e Iniciativas

Título	**URL**
Common Data Interchange Format (CDIF)	www.eigroup.org/cdif/
IDEF Family of Methods	www.idef.com
Object Management Group (OMG)	www.omg.org/uml
Rational UML Resource Center	www.rational.com/uml/index.jsp
Software Engineering Body of Knowledge	www.swebok.org
Software Engineering Institute	www.sei.cmu.edu *http://www.sei.cmu.edu/*
The Object Constraint Language (IBM)	www-4.ibm.com/software/ad/standards/ocl.html
The precise UML group	www.cs.york.ac.uk/puml
The precise UML group	www.cs.york.ac.uk/puml
UML Center (PLATINUM technology)	www.platinum.com/corp/uml/uml.htm
UML Revision Task Force (OMG)	www.celigent.com/omg/umlrtf
UML Zone (DevX)	www.uml-zone.com
Unified Modeling Language - Version 1.3, June 1999	www.rational.com/uml/resources/documentation
XMI, the XML Metadata Interchange Format (IBM)	www-4.ibm.com/software/ad/standards/xmi.html

Empresas e Links Relevantes

Título	**URL**
Catalysis	www.catalysis.org
Extreme Programming (Donovan Wells)	www.extremeprogramming.org
ICONIX Software Engineering	www.iconixsw.com
Methodology.org	www.geocities.com/itmweb
Objects by Design	www.objectsbydesign.com
Ratio Group	www.ratio.co.uk

Leituras Recomendadas

Título	URL
Agent UML Web Site	aot.ce.unipr.it/auml
Applying the Unified Modeling Language (Sinan Si Alhir)	home.earthlink.net/~salhir/applyingtheuml.html
Arquitectura de Software	www2.umassd.edu/SECenter/SAResources.html
CASE Tool Index (David Alex Lamb)	www.qucis.queensu.ca/Software-Engineering/tools.html
CASE Tool Information (David Alex Lamb)	www.qucis.queensu.ca/Software-Engineering/case.html
Data Modeling	www.datamodel.org
Description of the UML with summary of its diagrams	www.advancedsw.com/what_is_uml.html
Guide to selecting a UML modeling tool	www.objectsbydesign.com/tools/modeling_tools.html
Introduction to CASE (Simon Stobart)	osiris.sunderland.ac.uk/sst/case2/welcome.html
Mapping Objects to Relational Databases	www-4.ibm.com/software/developer/library/mapping-to-rdb/
Object Modeling Methodologies	www.yy.cs.keio.ac.jp/~suzuki/object/method.html
Object Oriented Analysis and Design Site	www.voicenet.com/~reubenx
Object Oriented Analysis and Design Using UML	www.ratio.co.uk/white.html
Object-Oriented Software Architecture	www.scis.nova.edu/~fangchih
OO Analysis and Design Methods - A Comparative Review (University of Twente)	wwwis.cs.utwente.nl:8080/dmrg/OODOC/oodoc/oo.html
OO Analysis and Design Using UML (Ratio Group)	www.ratio.co.uk/white.html
RUP (Rational Unified Process)	www.rational.com/products/rup/index.jsp
Selection list of UML tools.(Objects by Design)	www.objectsbydesign.com/tools/umltools_byCompany.html
Standardization and the UML (Martin Fowler)	www2.awl.com/cseng/titles/0-201-89542-0/techniques/standards.htm
Techniques for Object Oriented Analysis and Design	www2.awl.com/cseng/titles/0-201-89542-0/techniques
The Case Tool Home Page	osiris.sunderland.ac.uk/sst/case2/welcome.ht

	ml
The Object-Oriented Advantage	www.firststep.com.au/education/solid_ground/oo_dev.html
The Object-Oriented Page	www.well.com/user/ritchie/oo.html
Transforming XMI to HTML	www.objectsbydesign.com/projects/xmi_to_html.html
UML dictionary	softdocwiz.com/UML.htm
UML glossary	www.essaim.univ-mulhouse.fr/uml/documentation/version1.0/glossary/glossary.html
UML Modeling Tools	http://www.mrcase.net/education/UML/Acticle/tools/umltools_byCompany.html
Using Object Modeling CASE Tools (DBMS and Internet Systems)	www.dbmsmag.com/9707d16.html
What Is a Methodology? (The Object Agency)	www.toa.com/pub/methodology_article.txt
What is Object-Oriented Software?	catalog.com/softinfo/objects.html
What is the Unified Modeling Language (Sinan Si Alhir)	home.earthlink.net/~salhir/whatistheuml.html
Why Use Analysis and Design Techniques?	www2.awl.com/cseng/titles/0-201-89542-0/techniques/whyUse.htm

Catálogos de Informação

Título	URL
CETUS Links: Links de Orientação por Objectos	mini.net/cetus/software.html
CETUS Links: Lista de Ferramentas OO	mini.net/cetus/oo_ooa_ood_tools.html
CETUS Links: Object-Oriented Analysis & Design Methods	www.cetus-links.org/oo_ooa_ood_methods.html
CETUS Links: Object-Oriented Analysis & Design Tools	www.cetus-links.org/oo_ooa_ood_tools.html
CETUS Links: UML : Books	www.cetus-links.org/oo_uml.html#oo_uml_books
CETUS Links: Unified Modeling Language (UML)	www.cetus-links.org/oo_uml.html
Links de Engenharia de Software	www.enteract.com/~bradapp/links
Rational UML Resource Center	www.rational.com/uml/index.jsp
Rational Unified Process Whitepapers	www.rational.com/products/rup/whitepapers.jsp
Rational Whitepapers	www.rational.com/products/whitepapers/index.jsp
Recommended reading (Rational Software)	www.rational.com/uml/reading/index.jsp
The UML Bibliography (Mark Richters)	www.db.informatik.uni-bremen.de/umlbib/home.html
UML Central	www.advancedsw.com/uml_central.html

Ferramentas CASE

Empresa	Ferramenta	URL
Adaptive Arts	Simply Objects Modeler	www.adaptive-arts.com
Advanced Software Technologies (adquirida pela Embarcadero Technologies)	GDPro	www.advancedsw.com
Aonix	Software through Pictures	www.aonix.com/content/index.html
Artisan	Real Time Modeler	www.artisansw.com
Casewise	Casewise Corporate Modeler	www.casewise.com
Computer Associates	Paradigm Plus, Erwin, Bpwin, Cool Application Development Solutions	www.cai.com
Embarcadero Technologies	ER/Studio, DBArtisan	www.embarcadero.com
IDS Scheer AG	Aris	www.ids-scheer.com
Mega International	Mega Suite	www.mega.com
Microgold Software	WithClass	www.microgold.com
Microsoft	Visual Modeler	www.microsoft.com
Oracle	Designer, Developer	www.oracle.com
Popkin Software	System Architect	www.popkin.com
Princeton Softech	Select / Enterprise	www.princetonsoftech.com/index.asp
Proforma	ProVision	www.proformacorp.com
Rational	Rose, Rational Suite Product Family	www.rational.com
Silverrun Technologies	Silverrun	www.silverrun.com
Softeam	Objecteering	www.objecteering.com/us/index.htm
Softera	SoftModeler	www.softera.com
Sybase	PowerDesigner	www.sybase.com
Telelogic	ObjectGeode, Telelogic Tau UML Suite	www.telelogic.com
Visible Systems Corporation	Visible Analyst	www.visible.com
Visual Object Modelers	Visual UML	www.visualobject.com

Apêndice B – Glossário, Siglas e Abreviaturas

Neste apêndice apresenta-se separadamente três tabelas com informação do glossário, das siglas, e das abreviaturas adoptadas ao longo de todo o livro.

O glossário apresenta a tradução de um conjunto de termos e expressões utilizadas neste livro, da língua inglesa para a portuguesa. O objectivo é de esclarecer o leitor, porventura mais habituado com o vocábulo inglês, de algumas das traduções realizadas neste livro.

Para uma descrição mais alargada de termos não descritos na Secção B.1 consulte-se, por exemplo, a referência *http://berlin.inesc.pt/gtti/gtti.html*, a qual é o resultado de um esforço colectivo de alguns investigadores portugueses. Para uma consulta mais completa do significado de termos definidos no âmbito do UML e no âmbito de processos de desenvolvimento de software consulte-se na Web, por exemplo, as referências "*UML Dictionary*" (*http://softdocwiz.com/UML.htm*) ou "*UML Glossary*" (*www.essaim.univ-mulhouse.fr/uml/documentation/version1.0/glossary/glossary.html*).

B.1 Glossário

Inglês	Portugês
application programming interface (API)	interface de programação
boundary	fronteira ou interface
callback	método invertido
classifier	classificador
constraint	restrição
deployment	instalação
design pattern	padrão de desenho
embedded	embutido
extend	extensão
extension point	ponto de extensão
factory method	método-fábrica
framework	infra-estrutura (software) ou quadro de referência
include	inclusão
input event	evento de recepção
interface	interface
Internet	Internet
link	ligação
middleware	middleware
multithread	multiactividade
output event	evento de envio/emissão
package	pacote, biblioteca
plug-in	plug-in
qualifier	qualificador
remote method invocation (RMI)	invocação de método remoto
scripting language	linguagem de scripting
signal	sinal
stereotype	estereótipo
swimlanes	pistas

tag value	marca com valor
template	molde
thread	actividade, fio de execução
tier	camada, nível
use-case	caso de utilização
workflow	tarefa (ou workflow)

B.2 Siglas mais Usadas

Sigla	Expansão
API	Application Programming Interface
BD	Base de Dados
CASE	Computer Aided Software Engineering
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
GTTI	Glossário de Termos Técnicos Informáticos
GUI	Graphic User Interface
HTTP	Hypertext Transport Protocol
HTML	Hypertext Markup Language
IDL	Interface Definition Language
JDBC	Java Data Base Connectiviy
JDK	Java Development Kit
OCL	Object Constraint Language
ODBC	Open DataBase Connectivity
OMG	Object Management Group
OMT	Object Modeling Language
OO	Orientado por objectos / baseado em objectos
OOSE	Object Oriented Software Engineering
ORB	Object Request Broker
RMI	Remote Method Invocation
RUP	Rational Unified Process
SGBD	Sistema de Gestão de Bases de Dados
SQL	Standard Query Language
UML	Unified Modeling Language
URL	Uniform Resource Locator
WAE	Web Application Extension
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XP	Extreme Programming

B.3 Abreviaturas

Abreviatura	Significado(s)
vid.	veja-se, ver
i.e.	isto é, por conseguinte
e.g.	por exemplo
etc.	etecetera, outros
et al.	e outros (autores)
vs.	versus, por comparação com

REFERÊNCIAS

[Alexander77] C. Alexander, S. Ishikawa, M. Silverstein et al. *A Pattern Language*. Oxford University Press, 1977.

[Anthony65] R. N. Anthony. *Planning and Control: A Framework for Analysis*. Harvard University Press, 1965.

[Archer86] James Archer, Michael Devlin. *Rational's Experience Using Ada for Very Large Systems.* Proceedings of the First International on Ada, Junho 1986.

[Bach96] Volker Bach. *Selecting Software Tools for Business Process Redesign*. Proceedings of the 4th European Conference on Information Systems, Julho 1996.

[Banker91] R. D. Banker, R. J. Kauffman. *Reuse and Productivity in Integrated Computer-Aided Software Engineering: An Empirical Study*. MIS Quarterly, vol. 15, no. 3, Setembro 1991.

[Beck00] Kent Beck, Martin Fowler. *Planning Extreme Programming*. Addison-Wesley, 2000.

[Beck99] Kent Beck. *EXtreme Programming EXplained*. Longman Higher Education, 1999.

[Berard93] E. V. Berard. *Essays on Object-Oriented Software Engineering*. Addison-Wesley, 1993.

[Berner99] S. Berner, G. Martin, S. Joos. *A Classification of Stereotypes for Object-Oriented Modeling Languages*. *Proceedings of UML'99*. Springer, 1999.

[Bikson84] T. K. Bikson, B. Gutek. *Implementation of Office Automation*. Rand Corporation, 1984.

[BirtWhistle79] G.M. Birtwhistle, O.J. Dahl, B. Myhrhaug, K. Nygaard. *Simula Begin*. Chartwell-Bratt, 1979.

[Block83] R. Block. *The Politics of Projects*. Yourdon Press, 1983.

[Blum94] Bruce Blum. *A Taxonomy of Software Development Menthods*. Communications of the ACM, Vol 37, Nº 11, Novembro 1994.

[Boar98] Bernard H. Boar. *Constructing Blueprints Enterprise IT Architectures*. John Willey & Sons, 1998.

[Boehm86] Barry Boehm. *A Spiral Model of Software Development and Enhancement*. IEEE Computer, vol. 21, no. 5, 1988, pp. 61-82.

[Boehm96] Barry Boehm. *Anchoring the Software Process*. IEEE Software, vol. 13, no. 4, Julho 1996, pp. 73-82.

[Bohem00] Barry Bohem, Ellis Harrowitz. *Software Cost Estimation with COCOMO II*. Prentice Hall, 2000.

[Bohm66] C. Bohm, G. Jacopini. *Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules*. Communications of the ACM, Vol. 11, No. 4, Agosto 1966.

[Booch00] Grady Booch. *Unifying Enterprise Development Teams with the UML*. Rational, 2000.

[Booch86] Grady Booch. *Object-Oriented Development*. IEEE Trans. Software Engineering, vol. 12, no. 2, Fevereiro 1986.

[Booch93] Grady Booch, Doug Bryan. *Software Engineering with Ada*, 3ª edição. Benjamin/Cummings, 1993.

[Booch93] Grady Booch, Doug Bryan. *Software Engineering with Ada.* Benjamin/Cummings, 1993.

[Booch94] Grady Booch. *Object-Oriented Analysis and Design with Applications*, 2ª edição. Addison Wesley, 1994.

[Booch95] Grady Booch. *Object Solutions*. Addison-Wesley, 1995.

[Booch99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guid*e. Addison Wesley, 1999.

[Brooks86] Frederick P. Brooks Jr.. *No Silver Bullet - Essence and Accidents of Software Engineering*. Information Processing '86. Elsevier Science Publishers B.V., 1986.

[Cassidy98] Anita Cassidy. *A Practical Guide to Information Systems Strategic Planning*. St. Lucie Press, 1998.

[Coad00] Peter Coad, Eric Lefebvre, Jeff de Luca. *Java Modeling In Color With UML: Enterprise Components and Process*. Prentice Hall, 2000.

[Coad91] Peter Coad, Edward Yourdon. *Object-Oriented Analysis,* 2ª edição. Yourdon Press, 1991.

[Cockburn00] Alistair Cockburn. *Writing Effective Use Cases* (The Crystal Collection for Software Professionals). Addison-Wesley, 2000.

[Conallen00] Jim Conallen. *Building Web Applications with UML*. Addison Wesley, 2000.

[Conallen99] Jim Conallen. *Modelling Web Applications Architectures with UML*. Communications of the ACM, vol. 42, no. 10, Outubro 1999.

[Courtois85] P. Courtois. *On Time and Space Decomposition of Complex Structures*. Communications of the ACM, vol. 28, nº 6, Junho 1985.

[DeMarco78] Tom DeMarco. *Structured Analysis and System Specification*, Yourdon Press, 1978.

[Dijkstra65] B. W. Dijkstra. *Programming Considered as a Human Activity*. Proceedings of the 1665 IFIP Congress, North Holland Publishing Company, 1965.

[Dijkstra69] B. W. Dijkstra. Structured Programming. NATO Conference on Software Engineering, Outubro 1969.

[Diller94] Antoni Diller. *Z: An Introduction to Formal Methods*, 2ª edição. Addison-Wesley, 1994.

[Dix98] Alan J. Dix et al. *Human-Computer Interaction*, 2ª edição. Prentice Hall, 1998.

[D0D80] US Department of Defense. *Reference Manual for the Ada Programming Language*. 1980.

[Dorling91] A. Dorling, P. Simms. *ImproveIT*. Ministry of Defense, London, 1991.

[Drummond96] H. Drummond. *Escalation in Decision Making: The Tragedy of Taurus*. Oxford University Press, 1996.

[Ellis90] M. Ellis, B. Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, 1990.

[Evans01] Gary Evans. *A Simplified Approach to RUP*. The Rational Edge, Janeiro 2000.

[Finlay94] P. N. Finlay, A. C. Mitchell. *Perceptions of the Benefits form the Introduction of CASE: An Empirical Study*. MIS Quarterly, vol. 18, no. 4, Dezembro 1994.

[Firesmith96] Donald Firesmith. *Object-Oriented Requirements Analysis and Logical Design: A Software Engineering Approach*. John Wiley & Sons, Inc., 1996.

[FK92] Robert Fichman, and Chris Kemerer. *Object-Oriented and Conventional Analysis and Design Methodological: Comparision and Critique*. IEEE Computer, Outubro 1992.

[Fowler00] Martin Fowler. *The New Methodology*. Software Development Magazine, Dezembro 2000.

[Fowler96] Martin Fowler. *Analysis Patterns : Reusable Object Models*. Addison-Wesley, 1996.

[Fowler99a] Martin Fowler, Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Languag*e, 2ª edição. Addison-Wesley, 1999.

[Fowler99b] Martin Fowler. *Refactoring*. Addison-Wesley Longman Publishing CO, 1999.

[Gamma94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[Gane82] Trish Gane, Chris Sarson. *Structured Systems Analysis*. McDonnell Douglas, 1982.

[General79] Comptroller-General. *Report to Congress*. FGMSD-80-4, 1979.

[Gibbs94] W. Gibbs. *Software Chronic Crisis*. Scientific American, 271, 3 Setembro. Pp 72-81.

[Goldberg89] Adele Goldberg. *Smalltalk-80*, 2ª edição. Addison-Wesley Longman Publishing CO, 1989.

[Gris96] M. Griss. *Domain Engineering And Variability In The Reuse-Driven Software Engineering Business*. Object Magazine, Dezembro 1996

[Harel87] D. Harel. *Statecharts: A Visual Formalism for Complex Systems*. Science of Computer Programming 8, (1987), pp. 231-274.

[Harel96] D. Harel and A. Naamad. *The STATEMATE Semantics of Statecharts*. ACM Trans. Soft. Eng. Method 5:4, October 1996.

[Hofmeister99] C. Hofmeister, R. Nord, D. Soni. *Applied Software Architecture*. Addison Wesley, 1999.

[Hsia98] John Hsia. *Rose Extensibility: An Architectural Overview*. Rose Architect, Outubro 1998.

[Iivari96] J. Iivari. *Why are CASE tools not used?*. Communications of the ACM, vol. 39, no. 10, Outubro 1996.

[ISO91] *ISO 9000-3. Guidelines for the Application of ISO 9001 to the Development, Supply and Maintenance of Software.* International Standardization Organization, 1991.

[ITU-T93] ITU-T, Recommendation Z.100. *Specification and Description Language (SDL)*, ITU-T, 1993.

[Jackson95] Michael Jackson. *Software Requirements and Specifications*. Addison-Wesley, 1995.

[Jacobson85] Ivar Jacobson. *Concepts for Modeling Large Real Time Systems.* The Royal Institute of Technology, 1985.

[Jacobson92] Ivar Jacobson et al. *Object-Oriented Software Engineering: A Use Case Approach*. Addison Wesley, 1992.

[Jacobson97] Ivar Jacobson, Martin Griss, Patrik Jonsson. Software Reuse – Architecture, Process and Organization for Business Success. Addison Wesley, 1997.

[Jacobson99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.

[Jayaratna94] N. Jayaratna. *Understanding and Evaluating Methodologies, NIMSAD: A Systemic Framework*. McGraw-Hill, 1994.

[Johnson95] J. Johnson. *Chaos: The Dollar Drain of IT Project Failures, Application Development Trends*, Jan. pp 41-7.

[Johnson95] J. Johnson. *Chaos: The Dollar Drain of IT Project Failures*. Application Development Trends, Janeiro 1995.

[Kobryn99] Cris Kobryn. *UML 2001: A Standardization Odyssey.* Communications of the ACM, Vol. 42, Nº 10, Outubro 1999.

[Kruchten00] Philippe Kruchten. *The Rational Unified Process* , 2ª edição. Addison Wesley, 2000.

[Kruchten95] Philippe Kruchten. *Architectural Blueprints - The 4+1 View Model of Software Architecture*. IEEE Software 12 (8), pp 42-50, Novembro 1995.

[Kulak00] D. Kulak, E. Guiney. *Use Cases, Requirements in Context*. Addison Wesley, 2000.

[Larman98] C. Larman. *Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design*. Prentice-Hall. 1998.

[Larsen99] Grant Larsen. *Designing Component-Based Frameworks Using Patterns in the UML.* Communications of the ACM, Vol. 42, Nº 10, Outubro 1999.

[Lederer88] A. L. Lederer, V. Sethi. *The Implementation of Strategic Information Systems Planning Methodologies*. MIS Quaterly, Setembro 1988.

[Lehman79] J. H. Lehman. *How Software Projects are Really Managed*. Datamation, Janeiro 1979.

[Leveson87] N. G. Leveson, J. L. Stolzy. *Safety Analysis Using Petri Nets*. IEEE Trans. on Software Engineering, Vol. 13, No. 3, Março 1987.

[Leveson93] N. G. Leveson, C. S. Turner. *An investigation of the Therac-25 Accidents*. IEEE Computer 26, Julho 1993, pp. 18-41.

[Malan96] R. Malan, D. Coleman, R Letsinger et al. *The Next Generation of Fusion.* Fusion Newsletter, Outubro 1996.

[Martin89] J. Martin. *Information Engineering: Introduction Vol. 1*. Prentice Hall, 1989.

[Martin92] J. Martin, J. J. Odell. *Object-Oriented Analysis and Design*. Prentice Hall, 1992.

[Martin95] James Martin, James Odell. *Object-Oriented Methods, A Foundatio*n, Prentice Hall, 1995.

[Martin96] James Martin, James Odell. *Object-Oriented Methods, Pragmatic Considerations*. Prentice Hall, 1996.

[McClure89] Carma McClure. *The CASE Experience*. Byte, Abril 1989.

[McConnell96] Steve McConnell. *Rapid Development: Taming Wild Software Schedules*. Microsoft Press, 1996.

[McFarlan83] F. W. McFarlan, J. L. McKenney. *Corporate Information Systems Management*. Irwin, 1983.

[Mellor94] P. Mellor. *CAD: Computer-Aided Disaster*. Technical Report, Centre for Software Reliability, City University, London, UK Julho 1994.

[Mensah91] K. Ewusi-Mensah, Z. H. Przasnyski. *On Information Systems Project Abandonment: An Exploratory Study of Organisational Practices*. MIS Quaterly, Março 1991.

[Muller00] Robert J. Muller. *Database Design for Smarties: Using UML for Data Modeling*. Morgan Kaufmann Publishers, 1999.

[Neumann80] P. G. Neumann. *Letter form the Editor*. ACM SIGSOFT Software Engineering Notes 5, Julho 1980, pp. 2.

[Nunes99] Nuno Nunes, et. Al. *Workshop on Interactive System Design and Object Models (WISDOM'99)*. Proceedings ECOOP'99 Workshops, Lecture Notes in Computer Science 1743, Springer, 1999.

[O'Brien00] James O'Brien. *Introduction to Information Systems, Essentials for the Internetworked Enterprise*, 9ª Edição. McGraw-Hill, 2000.

[Odell00] James Odell, H. Van Dyke Parunak, Bernhard Bauer. *Extending UML for Agents*. Proc. of the Agent-Oriented Information Systems Workshop at the 17th National Conference on Artificial Intelligence (AOIS Worshop at AAAI 2000), 2000.

[Odell97] James Odell, Guus Ramackers. *Toward a Formalization of OO Analysis.* Journal of Object-Oriented Programming, Vol 10, No 4, Junho 1997.

[OMG98] Object Managament Group, XMI Partners. *XML Metadata Interchange (XMI)*, v1.0, Outubro 1998.

[OMG99] Object Managament Group, UML Revision Task Force. *OMG Unified Modeling Language Specification*. Version 1.3, 1999.

[Orange97] Kathy Orange, Graham Orange, Lesley Semmens. *Structured Techniques for Systems Analysis*. McGraw-Hill, 1997.

[Orlikowski93] W. J. Orlikowski. *CASE tools as Organizational Change: Investigating incremental and radical changes in systems development* . MIS Quarterly, vol. 17, no. 3, Setembro 1993.

[Page-Jones80] Meiler Page-Jones. *The Practical Guide to Structured Systems Design*. Yourdon Press, 1980.

[Parnas72] D. L. Parnas. *On the Criteria to be Used in Decomposing Systems into Modules*. Communications of the ACM, Vol. 15, No. 12, Dezembro 1972.

[Paulk93] M. Paulk et al. *Capability Maturity Model for Software*. Software Engineering Institute, Carnegie Mellon University, 1993.

[Penker00] Magnus Penker, Hans-Erik Eriksson. *Business Modeling With UML: Business Patterns at Work*. Addison-Wesley, 2000.

[Phan95] D. D. Phan, D. R. Vogel. *Empirical Studies in Software Development Projects: Field Survey and OS/400 Study*. Information and Management, nº 28, 1995.

[Plauger96] P. Plauger, A. Stepanov, M. Lee, D. Musser. *The Standard Template Library*. Prentice Hall, 1996.

[Popkin00] Popkin Software & Systems. *Building an Enterprise Architecture - The Popkin Process*. 2000.

[Popkin99] Popkin Software & Systems. *System Architect 2001 Tutorial*. 1999.

[Pressman00] Roger Pressman. *Software Engineering - A Practitioner's Approach*. McGraw-Hill, 2000.

[Primozic90] Edward Primozic. *Strategic Choices: Supremacy, Survival or Sayonara*. McGraw-Hill, 1990.

[Proforma00] Proforma. *Enterprise Application Modeling – Vision and strategy for the ongoing development of ProVision Workbench*. A Whitepaper, 2000.

[Quatrani00] Tery Quatrani. *Visual Modeling with Rational Rose 2000 and UML*. Addison-Wesley. 2000.

[Ramackers95] Ramackers, G. and Clegg, D., *Object Business Modeling, requirements and approac,.* in Sutherland, J. and Patel, D. (eds.), Proceedings of the OOPSLA95 Workshop on Business Object Design and Implementation, Springer Verlag.

[Ramackers96] Ramackers, G. and Clegg, D.. *Extended Use Cases and Business Objects for BPR*. ObjectWorld UK '96, London, June 18-21, 1996.

[Rational00a] Rational. *The UML and Data Modeling*, A Whitepaper. 2000.

[Rational00b] Rational. *Mapping Object to Data Models with the UML*, A Whitepaper. 2000.

[Rational00c] Rational. *Using Data Modeler, Rational Rose 2001*. 2000.

[Rational00d] Rational. *Rational Rose 2000e, Using Rose Visual Basic*. 2000.

[Rational00e] Rational. *Rational Rose 2000e, Quick Start with Rose Visual Basic*. 2000.

[Rational00f] Rational. *Rational Rose 2000e, Using Rose Visual Basic*. 2000.

[Rational00g] Rational. *Introducing Rational Suite*. 2000.

[Rational00h] Rational. *Rational Rose 2000e, Using Rose Oracle8*. 2000.

[Rational00i] Rational. *Rational Rose 2000e, Rose Extensibility User's Guide*. 2000.

[Rational00j] Rational. *Rational Rose 2000e, Using Rose Extensibility User's Guide*. 2000.

[Rational98] Rational. *Rational Unified Process, Best Practices for Software Development Teams*. A Whitepaper, 1998.

[Rational99] Rational. *Building Web Solutions with the Rational Unified Process: Unifying the Creative Design Process and the Software Engineering Process*, A Whitepaper. 1999.

[Rosenberg99] Doug Rosenberg. *Use Case Driven Object Modeling with UML*. Addison-Wesley. 1999.

[Rumbaugh91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenzen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Rumbaugh99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manua*l. Addison Wesley, 1999.

[Schach99] Stephen Schach. *Classical and Object-Oriented Software Engineering with UML and Java*, 4ª edição. McGraw-Hill, 1999.

[Selic94] B. Selic, G. Gullekson, and P. Ward, *Real-Time Object-Oriented Modelin*g, John Wiley & Sons, 1994.

[Sellers90] Brian Henderson-Sellers, Julian Edwards. *The Object-Oriented Systems Life Cycle*. Communications of the ACM, Vol. 33, nº 9, Setembro 1990.

[Sharma00] Srinarayan Sharma, Arun Rai. *CASE Deployment in IS Organizations*. Communications of the ACM, Vol. 43, Nº 1, Janeiro 2000.

[Shlaer88] S. Shlaer, S. J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Yourdon Press, 1988.

[Shlaer96] Sally Shlaer, Neil Lang. *Shlaer-Mellor Method: The OOA96 Report,* versão 1.0. Project Technology, Inc., 1996.

[Silberschatz98] Abraham Silberschatz. *Database System Concepts*, 3ª edição. McGraw-Hill, 1998.

[Sommerville97] I. Sommerville, P. Sawer. *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, 1997.

[Souza99] D. D'Souza and A. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approac*h. Addison-Wesley, 1999.

[Sowa92] J. F. Sowa, J. A. Zachman. *Extending and Formalizing the Framework for Information Systems Architecture*. IBM Systems Journal, vol. 31, nº 3, 1992.

[Spivey88] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge University Press, 1988.

[Stevens74] W. P. Stevens, G. J. Myers, L. L. Constantine. *Structured Design*. IBM Systems Journal, Vol. 13, No. 2, Maio 1974.

[Stroustrup88] B. Stroustrup. *What is Object-Oriented Programming*. IEEE Software, vol. 5, no. 3, pp 10-20, Maio 1988.

[Taylor90] D. A. Taylor. *Object-Oriented Technology: A Manager's Guide*. Addison-Wesley, 1990.

[Terry90] B. Terry, D. Logee. *Terminology for Software Engineering and Computer-Aided Software Engineering*. Software Engineering Notes, Abril 1990.

[Ullman94] J. D. Ullman. *Elements of ML Programming*. Prentice Hall, 1994.

[Vessey92] I. Vessey, S. L. Jarvenpaa, N. Tractinsky. *Evaluation of Vendor Products: CASE Tools as Methodology Companions*. Communications of the ACM, vol. 38, no. 1, Janeiro 1995.

[Walmsley90] Susan Walmsley, Shirley Williams. *Basically Modula-2*. Chartwell-Bratt, 1990.

[Warmer99] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UM*L. Addison-Wesley, 1999.

[Weaver98] Philip Weaver, Nick Lambrou, Matthew Walkley. *Practical SSADM Version 4+*, 2ª edição. Financial Times Prentice Hall, 1998.

[Whitten94] Jeffrey Whitten, Lonnie Bentley, Victor Barlow. *Systems Analysis and Design Methods*. Irwin, 1994.

[Winblad90] Ann Winblad, Samuel Edwards, David King. *Object-Oriented Software*. Addison-Wesley Publishing Company, Inc, 1990.

[Wirth71] Niklaus Wirth. *Program Development by Stepwise Refinement*. Communications of the ACM, Vol. 14, No. 4, Abril 1971.

[Works87] M. Works. *Cost Justification and the New Technology*. "A Program Guide for the CIM Implementation, 1987.

[Wurman97] Richard Wurman. *Information Architects*. Palace Press International, 1997.

[Yourdon95] Edward Yourdon, Katharine Whitehead, Jim Thomann, Karin Oppel, Peter Nevermann. *Mainstream Objects - An Analysis and Design Approach for Business*. Yourdon Press, 1995.

[Yourdon96] Edward Yourdon. *Rise and Ressurection of the American Programmer*. Yourdon Press, 1996.

[Yourdon99] Edward Yourdon. *Modern Structured Analysis*. Prentice Hall, 1999.

[Zachman87] John Zachman. *A Framework for Information Systems Architecture*. IBM Systems Journal, vol. 26, nº 3, 1987.

[Zachman96] John Zachman. *Enterprise Architecture: The Issue of the Century*. Database Programming and Design Magazine, Março 1997.

ÍNDICE REMISSIVO